TP Algo en autonomie, LDD2 & L3 UPSaclay

Ce TP est à faire en binôme (les monômes sont autorisés).

Langage C

Le but de ce TP est de programmer quelques fonctionnalités en C, qui font manipuler explicitement récursivité, pointeurs, listes chaînées, passages par adresse, etc. Certaines nécessitent pas ailleurs une réflexion algorithmique.

Le C++ est interdit (je veux vous voir faire des passages par adresse de pointeurs)

Attention, C est très permissif voire pousse-au-crime. On peut facilement y programmer très salement. C'est à vous d'être propre. Des points peuvent être retirés pour codes illisibles :

- Faire du code lisible, bien structurer, bien présenter et aérer les programmes. Un bon code est compréhensible par quelqu'un qui ne connaît pas le langage dans lequel il est écrit.
- Commenter. Faire du qualitatif, pas du quantitatif. Ne pas paraphraser le code facile. Expliquer le code difficile.
- Réduire au strict minimum l'utilisation des variables globales.
- Distinguer proprement expressions et instructions, distinguer procédures et fonctions.
- Vous pouvez utiliser du "sucre syntaxique" :

```
#define ISNOT !=
#define NOT !
#define AND &&
#define OR ||
#define then
typedef enum { false, true }  bool;
```

La notation :

Le principe de notation est le suivant (pour les partiel, examen, projet) : Une note brute est donnée par exemple sur 34. Les points comptent pour 1 jusqu'à 10, un 8/34 fait un 8/20, puis pour moitié après 10, un 20/34 fait 15/20, puis pour quart après 26, un 30/34 fait 19/20.

Partiels et exams d'algo sont exigeants, de l'ordre de 40% des étudiants de L3 info n'y ont pas la moyenne. Par contre, le projet est noté généreusement et il est facile d'y avoir la moyenne et plus, même si vous n'abordez pas les questions difficiles.

Travailler le projet donne donc des points d'avance pour l'U.E., mais il permet aussi et surtout de s'entraîner pour les partiel et examen et aide à y avoir une note correcte. L'expérience montre que ceux qui ne travaillent pas le projet se ramassent aux partiel et examen.

Comment avoir une mauvaise note au projet ?

En fournissant du code non testé qui ne marche pas. La notation sur du code qui ne marche pas sera moins sévère si vous annoncez qu'il ne marche pas.

Pire, en fournissant du code qui ne compile pas.

Des points peuvent être enlevés pour "code pénible qui fait mal au crane" : code affreux, présentation affreuse (mal aéré, mal indenté, lignes à rallonge qui débordent de l'écran, etc.), code mal placé "caché".

Le pompage (copie du code d'un autre binôme) et le plagiat (copie d'un bout de code d'un projet d'une année passée) sont interdits et peuvent vous conduire en commission de discipline. En cas de pompage, il n'est pas possible de distinguer le pompeur et le pompé. Les sanctions s'appliqueront aux deux. Il est donc fortement recommandé de ne pas "prêter" son code à un autre binôme, sous peine de mauvaise surprise très désagréable (il y a eu des cas...)

La discussion est autorisée. Si vous êtes bloqué, vous pouvez demander de l'aide à un autre binôme mais vous devrez produire votre propre code à partir de l'explication reçue.

Nommages et rendus

Gardez les noms de fonctions de l'énoncé, le correcteur doit les retrouver avec un contrôle F. S'il croit que vous ne l'avez pas faite suite à un mauvais nommage, tant pis pour vous...

Vous rendrez votre code dans des fichiers portant le nom du binôme, par exemple DupontDurand. S'il y a risque d'homonymie, vous rajouterez l'initiale du prénom ou plus si nécessaire. S'il y a un Jean Dupont et un Jacques Dupont, ce pourra être JnDupont et JqDupont. Il y aura un suffixe pour chaque fichier.

Vous rendrez les fichiers suivants, le nommage suppose que vous êtes monôme et que vous vous appelez Dupont :

- Dupont1.c pour la partie 1
- Dupont2.c pour la partie 2 sauf Permutations et ListesZ
- Dupont2P.c pour Permutations
- Dupont2Z.c pour les listesZ
- Dupont3.c pour la partie 3

Le code sera rendu sur ecampus.

Le code sera rendu 2 fois :

- Le prérendu sera corrigé et commenté.

Il a pour premier objectif de vous signaler vos erreurs ou lourdeurs pour que vous puissiez les corriger avant le rendu définitif.

Il a pour deuxième objectif de vous signaler des erreurs qu'il serait préférable de ne pas refaire au partiel ou à l'examen.

Des pré-rendus en retard sont tolérés, mais vous prenez le risque de ne pas avoir les commentaires avant le partiel ou l'examen, ou très tard. Voire de ne jamais les avoir.

Les codes manifestement en friche ne seront pas commentés. Le but du prérendu n'est pas de nous faire débuger à votre place. Si vous avez du code en friche, prière de le signaler en commentaire que nous ne perdions pas de temps à le regarder.

Signalez par un commentaire tout autre code qui n'est pas à lire, par exemple code vétuste.

Si vous avez un problème avec une fonction, vous pouvez toujours le signaler en commentaire. Le correcteur décidera s'il vous donne ou non une indication.

Certaines fonctions ne seront pas commentées. Par exemple, Permutations et ListeZ ne seront pas regardées pour le prérendu.

Si vous ramez et n'avancez pas, il est conseillé de faire un prérendu quand même avec le peu que vous aurez fait. Les commentaires pourraient vous débloquer pour la suite, et si vous avez du mal, c'est justement que vous avez besoin que nous fassions des commentaires.

Le prérendu n'est pas noté. Il n'a aucune influence sur la note.

- Le rendu définitif sera noté. Un rendu définitif en retard induira un malus.

Dates de rendus :

- prérendu de la partie 1 le 16 octobre matin
- prérendu de la partie 2 le 19 octobre matin
- rendu définitif des parties 1 et 2 le 20 novembre à 8h
- prérendu de la partie 3 le 9 décembre matin
- rendu définitif de la partie 3 le 15 janvier à 8h

Deux fichiers Rosaz1.c et Rosaz2.c sont disponibles sur ecampus, à récupérer pour démarrer votre projet.

1 Quelques calculs simples

- Calculez e en utilisant la formule $e = \sum_{n=0}^{\infty} 1/n!$.

  Il est pertinent d'éviter de recalculer factorielle depuis le début à chaque itération.

  Vous ne sommerez pas jusqu'à l'infini...
  Il est pertinent de vous demander quand et comment vous arrêter.

- On définit la suite $y_0 = e - 1$, puis par récurrence $y_n = n\, y_{n-1} - 1$.

  Un matheux vous dira que cette suite tend vers 0. Si vous souhaitez le montrer, utilisez la formule $e = \sum_{n=0}^{\infty} 1/n!$ pour démontrer $1/n < y_n < 1/(n-1)$.

  Codez une procédure pour faire afficher les 30 premiers termes. Faites une version qui travaille avec un float, une version avec un double et une version avec un long double. Que constatez-vous ?

  Si vous souhaitez l'explication, considérez la suite définie par $z_0 = e - 1 + \epsilon$, puis par récurrence $z_n = n\, z_{n-1} - 1$, et trouvez ce que vaut $z_n - y_n$

- Implémentez les 11 versions de power qui sont numérotées dans la correction du TD1 (confer icelle sur ecampus)

  Pour les versions 1 et 2a, écrire des versions qui fonctionnent avec n positif ou négatif. Rappel : $(-2)^2 = 4$, $(-2)^3 = -8$, $(-2)^{-2} = 2^{-2} = 0.25$, $(-2)^{-3} = -0.125$, 0^{-1} plante, et 0^0 n'est pas vraiment défini (0 ou 1 ?)

  Calculez $(1 + 1/N)^N$ pour N des puissances de 10 de plus en plus grandes, 1.1^{10}, 1.01^{100}, 1.001^{1000}, etc. Vers quoi la suite $(1 + 1/N)^N$ semble-t-elle tendre ? Tester les 11 versions avec N de grosses puissances de 10, $N = 10^9$, $N = 10^{12}$, $N = 10^{15}$. Les diverses versions sont-elles capables de gérer d'aussi gros nombres ? Observez d'éventuelles limites, en temps et en espace. Votre compilateur semble-t-il effectuer l'optimisation du récursif terminal ?

- Implémentez plusieurs méthodes pour calculer la fonction d'Ackermann, Sont possibles les versions suivantes (les deux premières sont dans la correction du TD) :
  - Une version purement récursive
  - Une version récursive en m et itérative en n
  - Une version itérative en m et récursive en n
  - Une version purement itérative
  - Une version purement itérative moins gourmande en mémoire que la précédente.

  Calculez les premières valeurs de A(m,0) dont A(5,0). Que se passe-t-il quand on tente de calculer A(6,0) (tourne sans donner de résultat, stop out of memory, stop MAXINT ?), les diférentes versions ont-elles le même comportement ?

- La suite d'entiers $(x_n)_{n \in N}$ est définie par récurrence: $x_0 = 2$
  puis $\forall n \geq 1, x_n = x_{n-1} + ln2(x_{n-1})$.

  $ln2$ étant le log 2 entier (cf TD1 exo 10.1)

  On a donc $x_0 = 2$, $x_1 = 3$, $x_2 = 4$, $x_3 = 6$, $x_4 = 8$, $x_5 = 11$, $x_6 = 14$, $x_7 = 17$, $x_8 = 21$ ...

  Coder la fonction $X(n)$. Donner quatre versions :
  - une version itérative,
  - une version récursive sans utiliser de sous-fonctionnalité,
  - une version récursive terminale avec sous fonction
  - une version récursive terminale avec sous procédure

  Utilisez les quatre méthodes pour calculer X_{100}. Vous devez obtenir le résultat par les quatre méthodes.

2 Listes-Piles

2.1

- **UnPlusDeuxEgalTrois** qui prend en argument une liste et rend vrai ssi le troisième élément est égal à la somme du premier et du second. Les éléments inexistants seront supposés valoir 0. Exemples, la fonction rend vrai sur [3,5,8,4,29], [2,-2], [] et faux sur [2,3,7,1], [2,2]

- **Croissante** qui prend une liste L et rend vrai ssi elle est strictement croissante.

- **NombreMemePosition** qui prend en argument deux listes d'entiers et rend le nombre d'éléments identiques à la même position.
  Par exemple, NombreMemePosition([3,6,9,0,3,4,2,9,2] , [3,9,6,0,2,2,2,2,2,2,5]) rendra 4 (le 3 en position 1, le 0 en position 4, le 2 en position 7 et le 2 en position 9)
  Donnez quatre versions de cette fonction :
  • Une version récursive sans sous-fonctionnalité (et non terminale)
  • Une version itérative
  • Une version qui utilise une sous-fonction récursive terminale.
  • Une version qui utilise une sous-procédure récursive terminale.

- **FonctVireDernier** qui prend une liste et REND une nouvelle liste qui est identique à la liste argument sauf qu'il y manque le dernier élément. Si l'argument est vide, la fonction rend une liste vide. Faire une version récursive simple (non terminale) puis (plus technique) une version itérative. Ne faire qu'une seule passe, donc ne pas utiliser ProcVireDernier (qui se trouve dans le poly) ni les variantes de Miroir.

- **AjouteDevantPremierZero** qui prend une liste L et un entier x et ajoute x devant le premier 0 de L. S'il n'y a pas de 0 dans L, x est ajouté en fin de liste. Exemple, si L est [4,0,5,0,8,0,1] avant l'appel et $x = 7$ alors L est [4,7,0,5,0,8,0,1] après l'appel.

- **AjouteDevantDernierZero** qui prend une liste L et un entier x et ajoute x devant le dernier 0 de L. S'il n'y a pas de 0 dans L, x est ajouté en fin de liste. Exemple, si L est [4,0,5,0,8,0,1] avant l'appel et $x = 7$ alors L est [4,0,5,0,8,7,0,1] après l'appel. Toutes les versions doivent être en une seule passe. Faire une version récursive avec argument inout. Puis (plus délicat) une version récursive terminale et une version itérative.

- **Tic** procédure qui prend en argument une liste d'entiers et qui ajoute un 0 devant tous les termes non nuls qui précèdent le premier 0 (s'il n'y a pas de 0: devant tous les termes) et enlève les zéros consécutifs qui suivent. Exemple, [3,7,2,0,0,0,8,9,0,0,2,1] est transformé en [0,3,0,7,0,2,8,9,0,0,2,1]

2.2

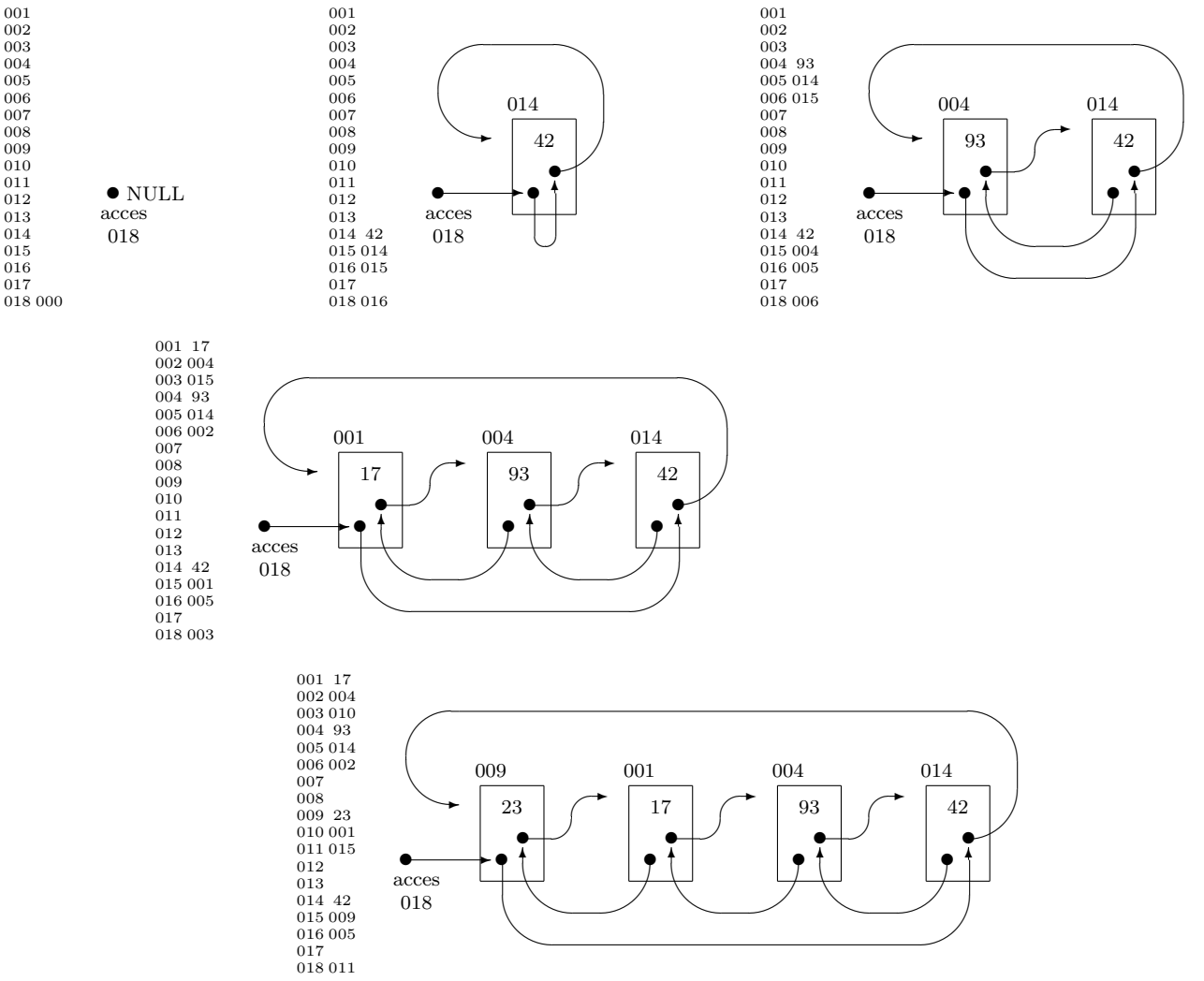
Cette partie ne sera pas lue dans le prérendu.

- Coder la fonction **Permutations** en utilisant la technique "diviser pour régner" du cours (confer le poly). Ajoutez des compteurs pour compter le nombre de malloc effectués (un pour chaque type). Observez la "compression" sur un type et la fuite mémoire sur l'autre. (Délicat :) Parvenez-vous à éliminer la fuite mémoire ? À compresser encore plus ?

- Les ListesZ sont construites avec des blocs. Chaque bloc possède trois champs, un champs valeur de type entier, un champs `next` qui pointe vers le bloc suivant, et un champs `prev` qui pointe vers le champs `next` du bloc précédent. S'il n'y a qu'un seul bloc, les blocs suivant et précédent sont le bloc lui-même. La structure est circulaire. L'ensemble est accessible par un pointeur `acces` qui pointe vers le champs `pred` d'un bloc. Si la listeZ est vide, le pointeur `acces` est vide.

  Écrire le code de la procédure `ZAjouteAvant` qui prend en argument un entier x et un pointeur `acces` et ajoute un nouveau bloc contenant x qui sera inséré juste avant le bloc à l'intérieur duquel `acces` pointe. Le pointeur `acces` pointera vers ce nouveau bloc après l'appel.

  Exemple, si la liste est vide, et que l'on appelle `ZAjouteAvant` avec 42, puis avec 93, puis avec 17, puis avec 23, nous obtiendrons successivement les configurations ci-dessous :



3 Arbres : Quadtrees

Les Quadtrees représentent des images en noir et blanc. Une image Quadtree est :

- soit blanche
- soit noire
- soit se décompose en 4 sous-images. haut-gauche, haut-droite, bas-gauche, bas-droite

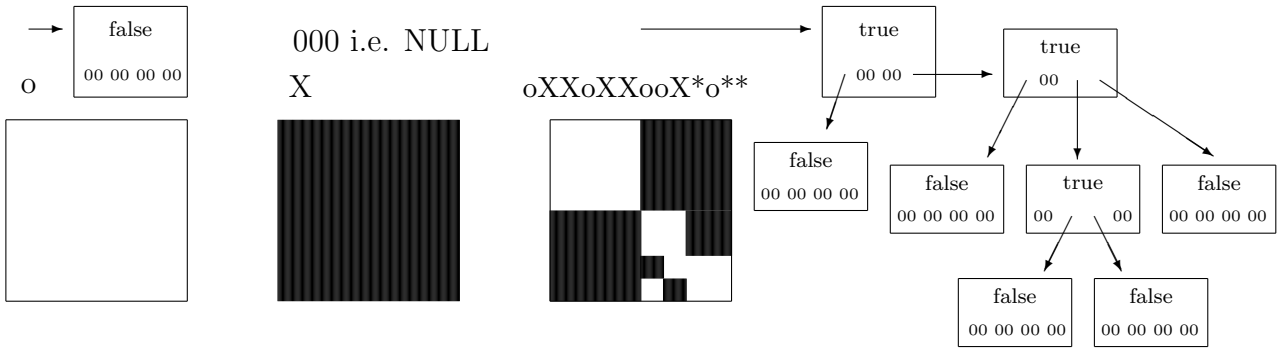
On représentera ces images avec la structure suivante :

```
typedef struct bloc_image
        { bool quatre ;
          struct bloc_image *hg, *hd, *bg, *bd ;
        } bloc_image ;
typedef bloc_image *image ;
```

Quand le pointeur est `NULL`, l'image est noire.

Quand il pointe vers un struct dont le champ `quatre` est `true`, l'image est obtenue en découpant l'image en 4, et en plaçant respectivement les images `hg, hd, bg, bd` en haut à gauche, en haut à droite, en bas à gauche, en bas à droite.

Quand il pointe vers un struct dont le champ `quatre` est `false`, l'image est blanche et les 4 champs `hg, hd, bg, bd` sont NULL.



3.1 Entrées Sorties

On utilisera la notation suivante pour les entrées sorties :

- o pour une image blanche
- X pour une image noire
- $x_1x_2x_3x_4*$ pour une image décomposée, avec x_1, x_2, x_3, x_4 les notations pour les sous images respectivement haut-gauche, haut-droite, bas-gauche, bas-droite.

Par exemple, l'écriture oooX*ooXo*oXoo*Xooo** représente un carré noir au centre de l'image.

Affichages :

- Le mode simple affiche une image selon le mode ci-dessus.
- En mode profondeur, la profondeur est donnée après chaque symbole.

  Par exemple, XooXo*oXXXoXoXX**oX** sera affiché comme suit :

  X1 o2 o2 X2 o2 *1 o1 X2 X3 X3 o3 X4 o4 X4 X4 *3 *2 o2 X2 *1 *0

Il est autorisé de rajouter des parenthèses à l'écriture (mais pas de les imposer à la lecture):
oooX*ooXo*oXoo*Xooo** peut s'écrire ((oooX*)(ooXo*)(oXoo*)(Xooo*)*)
oXXoX*XoooXoXoo**Xo** peut s'écrire (o(XXoX*)X(o(ooX(oXoo*)*)Xo*)*)

Lecture :

- Un point d'exclamation marquera la fin d'une entrée pour la lecture

Les caractères autres que oX*! sont sans signification et doivent être ignorés à la lecture.

3.2 Fonctionalités à écrire

Écrire les fonctions et procédures :

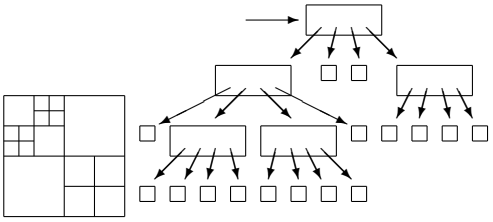
Rappel : le nom des fonctions doit être conservé, nous devons pouvoir retrouver rapidement une fonction par un contrôle F, et nous devons pouvoir injecter votre code dans notre programme sans avoir à modifier des noms.

1. Blanc() qui rend une image blanche à partir de rien.
   Noir() qui rend une image noire à partir de rien.
   Compose(i0,i1,i2,i3) qui construit une image composée des sous-images i0,...,i3.

2. Affichage(image)

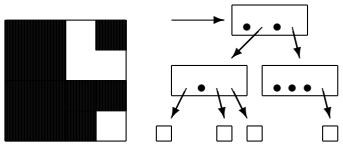
3. AffichageProfondeur(image)

4. Lecture() qui rend une image à partir des caractères tapés au clavier.
   Il est suggéré d'utiliser getchar() qui lit un caractère.

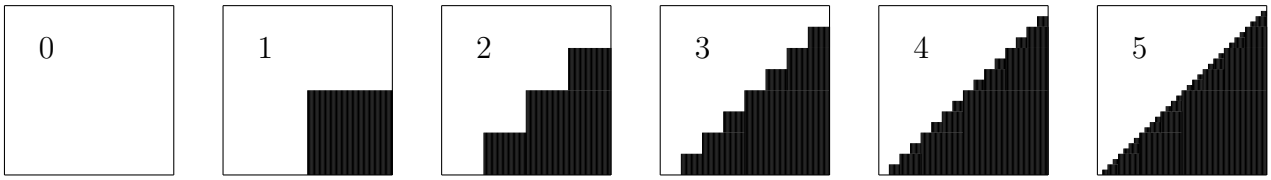


5. EstNoire(image) et EstBlanche(image) qui testent
   si l'image en argument est noire, resp. blanche.
   ooooo*oooo*o*oooooo** est blanche.
   // ((o(oooo*)(oooo*)o*)oo(oooo*)*)

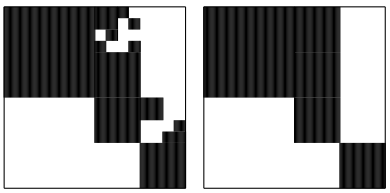
6. Aire(image) qui rend le taux de noir de l'image argument,
   Aire(XoXoo*XXXXo**)= 0.75
   // (X(oXoo*)X(XXXo*)*)



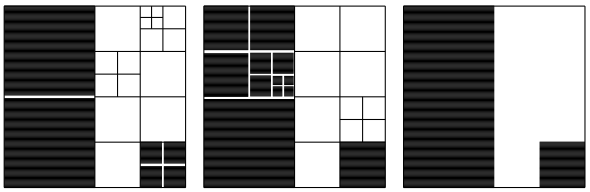
7. TriangleBD(int p) qui rend une image de profondeur p dans laquelle un pixel est noir ssi il est strictement en dessous de la diagonale bas-gauche haut-droit. TriangleBD(0) rendra o , triangleBD(1) rendra oooX*, triangleBD(3) rendra oooooX*oooX*X*ooooX*oooX*X*X*
   // (o(o(oooX*)(oooX*)X*)(o(oooX*)(oooX*)X*)X*)

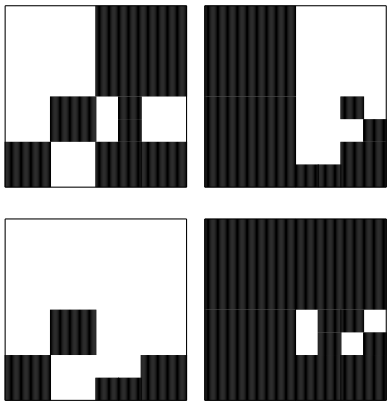


8. Arrondit(inout image, int p) qui arrondit l'image à profondeur p,
   i.e. aux pixels de dimension $1/2^p * 1/2^p$
   `Arrondit`(image,0) transforme une image en blanc ou noir,
   `Arrondit`(XXXooX*oXXo*oooX**oXo*oXXoooXXX**oX** ,2)
   arrondit l'image en XXoXo*oXooX**
   // (X([X(XooX*)(oXXo*)(oooX*)*]oXo*)o(X[Xoo(oXXX*)*]oX*)*)
   // (X(XoXo*)o(XooX*)*)



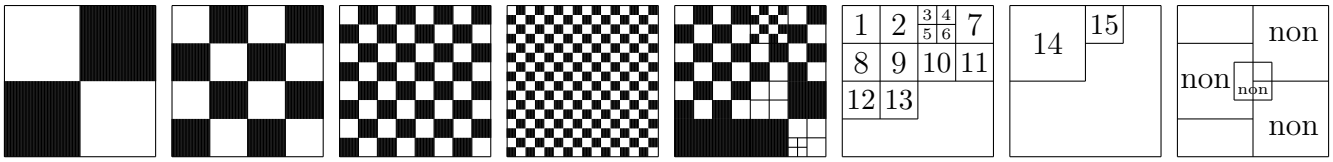
9. MemeDessin(image1, image2) qui teste si
   les deux images représentent le même dessin.
   Xooooo*ooo*oooo*o*XoooXXXX*** et
   XXXXXXXXXX***oooo*Xooooo*oX**
   ont le même dessin.
   // (X(o((oooo*)ooo*)(oooo*)o*)X(ooo(XXXX*)*)*)
   // ((XXX(XXX(XXXX*)*)*)(oooo*)X(o(oooo*)oX*)*)
   Ne pas simplifier les images en argument, ni des copies





10. InterUnion(inout image1, inout image2) qui transforme image1 en l'intersection des deux images, et image2 en l'union des deux images. Par exemple, si avant l'appel, image1 est oXoXXo*oXoX*oXX** et image2 est XoXoXooX*ooXX*X** , alors après l'appel, image1 est oooXXo*ooooXX*X** et image2 est XXXoXoX*XooX*XX**
// (oX(oXXo*)((oXoX*)oXX*)*),
// (XoX(o(XooX*)(ooXX*)X*)*),
// (oo(oXXo*)(oo(ooXX*)X*)*),
// (XXX((oXoX*)(XooX*)XX*)*)

11. CompteDamiers(image) qui rend le nombre de sous-arbres qui sont des damiers.
Il n'y a pas de damier de hauteur 0. oXXo* est le damier de hauteur 1.
Si A est le damier de hauteur h, alors AAAA* est le damier de hauteur h+1.
Les damiers de hauteur 2 et 3 sont ((oXXo*)(oXXo*)(oXXo*)(oXXo*)*) et
oXXo*oXXo*oXXo*oXXo**oXXo*oXXo*oXXo*oXXo**oXXo*oXXo*oXXo*oXXo**oXXo*oXXo*XXo*oXXo***
CompteDamier de l'image ci-dessous rend 15 : 13 damiers de hauteur 1 et 2 de hauteur 2.
(((oXXo*)(oXXo*)(oXXo*)(oXXo*)*)(((oXXo*)(oXXo*)(oXXo*)(oXXo*)*) (oXXo*)(oXXo*)(oXXo*)*)((oXXo*)(oXXo*)XX*)((oooo*)XX(oo(oooo*)o*)*)*)
L'image milieu-gauche ne compte pas, le dessin est celui d'un damier, mais ne correspond pas à un sous arbre. Itou pour le petit carré au centre de l'image. Le sous-arbre haut-droit n'est pas un damier bien que ses fils soient des damiers. Le sous-arbre bas-droit n'est pas un damier, le dessin est celui d'un damier mais l'arbre n'est pas simplifié.

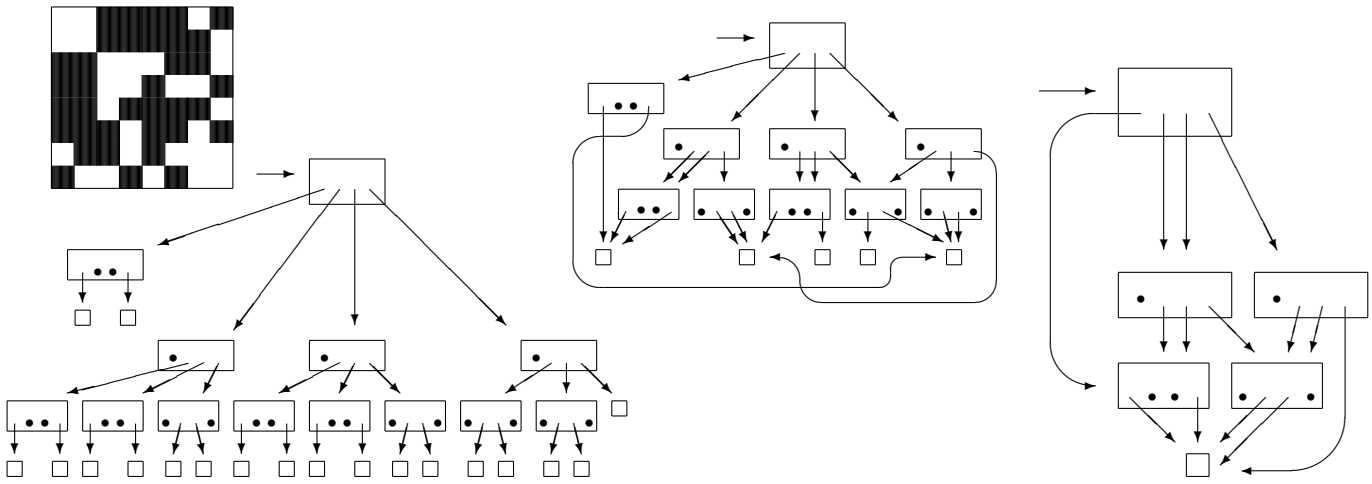


12. Compresse(inout image) qui transforme une image arbre en sa version DAG totalement compressée et Dilate(inout image) qui transforme une image DAG partiellement compressée en sa version arbre non compressée.

Un DAG (Directed Acyclic Graph) ressemble à un arbre mais un même `struc` peut servir plusieurs fois, ce qui permet d'économiser de l'espace mémoire. On parlera d'images partiellement compressées. Un DAG est totalement compressé ssi l'espace mémoire utilisé est minimmal.

Ci-dessous, une représentation par un arbre (qui est un DAG...), par un DAG partiellement compress; et par un DAG totalement compressé, de l'image

(oXXo*)(X(oXXo*)(oXXo*)(XooX*)*)(X(oXXo*)(oXXo*)(XooX*)*)(X(XooX*)(XooX*)o*)*



4 Appendice : la suite Yn

On définit la suite $y_0 = e - 1$, puis par récurrence $y_n = n\,y_{n-1} - 1$.

La suite semble tendre vers 0, puis au bout d'un certain moment, elle part vers plus ou moins l'infini.

Que se passe-t-il ?

Étude mathématique : On a $e = \sum_{n=0}^{\infty} 1/n!$, donc

$$
\begin{array}{lcccccccccccccc}
y_0 & = & \frac{1}{1} & + & \frac{1}{2} & + & \frac{1}{2*3} & + & \frac{1}{2*3*4} & + & \frac{1}{2*3*4*5} & + & \frac{1}{2*3*4*5*6} & + & \frac{1}{2*3*4*5*6*7} & + & \ldots \\
y_1 & = & & & \frac{1}{2} & + & \frac{1}{2*3} & + & \frac{1}{2*3*4} & + & \frac{1}{2*3*4*5} & + & \frac{1}{2*3*4*5*6} & + & \frac{1}{2*3*4*5*6*7} & + & \ldots \\
y_2 & = & & & & & \frac{1}{3} & + & \frac{1}{3*4} & + & \frac{1}{3*4*5} & + & \frac{1}{3*4*5*6} & + & \frac{1}{3*4*5*6*7} & + & \ldots \\
y_3 & = & & & & & & & \frac{1}{4} & + & \frac{1}{4*5} & + & \frac{1}{4*5*6} & + & \frac{1}{4*5*6*7} & + & \ldots \\
y_4 & = & & & & & & & & & \frac{1}{5} & + & \frac{1}{5*6} & + & \frac{1}{5*6*7} & + & \ldots \\
y_5 & = & & & & & & & & & & & \frac{1}{6} & + & \frac{1}{6*7} & + & \ldots
\end{array}
$$

Par récurrence, on a :

$y_n = \sum_{p>n} \frac{1}{p!/n!} = \frac{1}{(n+1)} + \frac{1}{(n+1)(n+2)} + \frac{1}{(n+1)(n+2)(n+3)} + \frac{1}{(n+1)(n+2)(n+3)(n+4)} + \ldots$

On en déduit que y_n est plus grand que $1/(n+1)$, le premier terme de la somme.

Par ailleurs, on peut minorer $(n+2)$, $(n+3)$, ... $(n+i)$ par $(n+1)$ et donc majorer $1/(n+2)$, $1/(n+3)$, ... $1/(n+i)$ par $1/(n+1)$.

On a donc

$y_n = \sum_{p>n} \frac{1}{p!/n!} < \frac{1}{(n+1)} + \frac{1}{(n+1)^2} + \frac{1}{(n+1)^3} + \frac{1}{(n+1)^4} + \ldots = \frac{1/(n+1)}{1-1/(n+1)} = 1/n$

Bref $\frac{1}{n+1} < y_n < \frac{1}{n}$, donc la suite est décroissante (car $y_{n+1} < \frac{1}{n+1} < y_n$) et elle tend vers 0 (car $y_n < \frac{1}{n}$).

Étude informatique :

Si on programme la suite, on ne peut pas initialiser y_0 à $e - 1$ avec son infinité de chiffres après la virgule. Ce sera une approximation par un flottant. Donc le vrai départ sera à $z_0 = e - 1 + \epsilon$ avec ϵ de l'ordre de 10^{-10} ou 10^{-20}, puis on fera $z_n = n z_{n-1} - 1$.

On a $z_n - y_n = n(z_{n-1} - y_{n-1})$, soit par récurrence, $z_n - y_n = n! * \epsilon$, soit $z_n = y_n + n! * \epsilon$.

Mais factorielle croit très vite et l'erreur de départ est rapidement amplifiée et finit par complètement prendre le dessus sur la valeur mathématique. Sur mon ordi, l'affichage cesse de décroître à n=9 (float), n=16 (double), n=19 (long double) puis part vers l'infini.

Vous pourriez tenter d'utiliser plus d'octets, mais le phénomène arrivera de toutes façon assez rapidement. Si vous aviez des superfloat sur 100 octets, le phénomène serait observé à partir de environ Y_{120}.
Avec 1000 octets, à partir de environ Y_{800}.
Avec 10000 octets, à partir de environ Y_{6000}.
Avec 100000 octets, à partir de environ Y_{48000}.
Avec 1000000 octets, à partir de environ Y_{400000}.