# TP Algo en autonomie, LDD2 & L3 UPSaclay

Ce TP est à faire en binôme. Les monômes sont autorisés.

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

Le C++ est interdit (les passages par adresse de pointeurs sont obligatoires)

**La notation :**

Le principe est le suivant (pour les partiel, examen et projet) : Une note brute est donnée par exemple sur 35. Les points comptent pour 1 jusqu'à 10, un 8/35 fait donc 8/20, puis pour moitié après 10, un 20/35 fait donc 15/20, puis pour un quart après 25, un 31/35 fait donc 19/20.

Partiels et exams d'algo sont longs et restent exigeants, avec des questions faciles, des questions médianes et des questions difficiles. 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 ne faites que les questions faciles et n'abordez pas les questions difficiles, 90% des L3 info y ont la moyenne. In fine, de l'ordre de 70% des L3 info ont la moyenne au module.

Travailler le projet donne donc des points d'avance pour l'U.E., surtout pour les étudiants faibles ou médians. Mais il leur permet aussi et surtout de s'entraîner pour les partiel et examen et aide à y avoir une note correcte. Tous les ans, des étudiants ne travaillent pas vraiment leur projet, tirent tout de même une note potable sur icelui, mais se ramassent méchamment aux partiel et exam, et in fine tirent une note d'U.E. médiocre.

Comment avoir une mauvaise note au projet ?

En fournissant du code non testé qui ne marche pas.

Pire, en fournissant du code qui ne compile pas.

Les tests ne sont pas demandés, mais la notation est sévère si un bug traîne qui aurait dû être repéré par des tests. La notation sera moins sévère si vous annoncez que le code ne marche pas.

Des points peuvent être enlevés si vous nous faites mal au crane : code affreux, présentation affreuse, code mal placé "caché".

Gardez les noms de fonctions de l'énoncé.

Le correcteur doit pouvoir 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...

Il doit aussi pouvoir les copier-coller dans son programme de tests sans avoir à tout renommer.

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.

• Quand l'énoncé dit "observez", "constatez", il n'est pas demandé de nous dire ce que vous avez observé. Mais faites-le tout de même. Parfois, c'est juste pour votre apprentissage. Parfois, c'est pour que vous détectiez un éventuel problème de votre code.

**Nommages et rendus**

Au début de vos fichiers, vous mettrez en commentaire les emails des deux membres du binôme.

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 2bis et 2ter
- Dupont2bis.c et Dupont2ter.c pour les parties 2bis et 2ter
- Dupont3.c pour la partie 3

Le code sera rendu sur ecampus. Un seul rendu par binôme (Vérifiez que votre binôme a bien déposé le projet), nous n'avons pas le temps de corriger votre projet deux fois, ni de partir à la pêche aux codes doublement rendus.

Vous rendrez le code 2 fois :

- Le pré-rendu sera corrigé et commenté. Objectifs :

Pédagogique : Vous apprendre à mieux programmer.

Note de projet : Vous signaler erreurs ou lourdeurs pour que vous puissiez les corriger avant le rendu définitif noté.

Notes de partiel et examen : 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 pré-rendu 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. Si vous avez un problème avec une fonctionnalité, vous pouvez le signaler en commentaire. Le correcteur décidera s'il vous donne ou non une indication.

Certaines fonctions ne seront pas commentées. Par exemple, les parties 2bis et 2ter ne seront pas regardées pour les pré-rendus.

Si vous ramez et n'avancez pas, il est conseillé de faire un pré-rendu 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. Il est plus pertinent pour vous que pour ceux qui avancent facilement.

Pour les projets rendus en retard, il sera donné priorité avant examens aux projets qui ont manifestement besoin de retour. Nous trouverions aimable que ceux qui avancent bien rendent le pré-rendu tôt pour nous laisser le temps de gérer sur le tard les prérendus de ceux qui rament.

Le pré-rendu n'est pas noté. Il n'a aucune influence sur la note.

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

**Dates de rendus :**

- pré-rendu de la partie 1 le 16 octobre matin
- pré-rendu de la partie 2.1 le 18 octobre matin
- rendu définitif des parties 1 et 2 le 1er décembre à 8h
- pré-rendu de la partie 3 le 13 décembre matin
- rendu définitif de la partie 3 le 12 janvier à 8h

Deux fichiers Algo1.c et Algo2.c sont disponibles sur ecampus, pour démarrer votre projet.

**Langage C**

Attention, C est très permissif voire pousse-au-crime. On peut facilement y programmer très salement. Il y a même une compétition du "code C le plus obscur" (https://www.ioccc.org).

C'est à vous d'être propre. Nous aurons 130 codes à lire. Nous retirerons des points pour codes sales.

• Faites du code lisible. Un bon code est compréhensible par quelqu'un qui ne connaît pas le langage dans lequel il est écrit.

• Aérez, structurez, présentez proprement, évitez les longues lignes qui débordent, indentez, mettez des /******/ souvent. Bref, facilitez la lecture

• Commentez. Faites des commentaires qualitatifs plutôt que quantitatif. Bof pour ceux qui paraphrasent le code évident et se taisent quand le code est difficile.

• Réduisez au strict minimum l'utilisation des variables globales.

• Distinguez proprement expressions et instructions, distinguez procédures et fonctions.

• C étant C, un certain nombre de codes bugés ne devraient pas compiler mais compilent quand même, et font n'importe quoi. `if (l->suite == 0)`, `if (l->valeur == NULL)` compilent, parce que le compilateur identifie pointeurs `(l->suite, NULL)` et entiers `(0, l->valeur)`, mais avec warning. Un warning, c'est une erreur. Et le code n'ayant pas été testé, il est sans valeur.

• N'utilisez pas de **Undefined Behavior**, ni de **comportements non standards**. Sinon votre code compilera et fonctionnera chez vous mais pas chez moi ! :

•• L'encapsulation, ou fonctions imbriquées, n'est pas dans la norme C, certains compilateurs l'acceptent, mais c'est très loin d'être la majorité. Le mien ne l'accepte pas.

•• L'appel de fonctions définies plus tard sans prédéclaration, `int f() { g();} int g() {}` n'est pas dans la norme C, quelques compilateurs l'acceptent, mais c'est loin d'être systématique. Le mien l'accepte mais avec de gros warnings.

•• La déclaration d'un tableau avec une taille non constante `int T[n]` est acceptée par plusieurs compilateurs, mais pas par tous. Ce n'est pas dans la norme.

•• Certains compilateurs mettent 0 par défaut dans les variables déclarées, mais d'autres laissent ce qui y traîne. Plusieurs codes rendus les années passées ne marchent pas chez moi pour cette raison.

•• Quand vous oubliez de faire un "return", certains compilateurs renvoient parfois la dernière valeur calculée (qui est sur le sommet de pile de calculs), et vous avez parfois la chance que ce soit justement ce qu'il fallait rendre. Il se peut donc que cela marche chez vous si vous oubliez le mot return ("`AppelRec(blabla);`" au lieu de "`return AppelRec(blabla);`") ou que vous oubliez le return ("`x=0; if b x++;`" au lieu de "`x=0; if b x++; return(x);`"). Une fois encore, cela ne marchera plus chez le voisin.

• Faites les passages par adresse correctement, pas comme dans le dernier exo du TD3.

• Évitez les petites maladresses :

Si test alors rendre vrai sinon rendre faux `->` rendre test

Si non test alors long sinon court `->` si test alors court sinon long

Si/TantQue/Rendre b == vrai `->` Si/TantQue/Rendre b

• Évitez les variables inutiles :

tmp = truc(x) ; x = tmp ; `->` x = truc(x) ;

tmp = truc() ; rendre tmp ; `->` rendre truc() ;

• Évitez les redondances de tests : if L == NULL ... else if (L != NULL) et Blabla

• N'hésitez pas à utiliser les options paranoïaques des compilateurs : -pedantic -Wall -Wextra

# 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... Quand et comment vous arrêterez-vous ?
  Faire trois versions qui rendent respectivement un `float`, un `double` et un `long double`.

- On définit la suite $y_0 = e - 1$, puis par récurrence $y_n = n\, y_{n-1} - 1$.
  Codez une procédure qui fait afficher les 30 premiers termes. Faites trois versions qui travaillent respectivement avec un `float`, un `double` et un `long double`. Que constatez-vous ?
  Un matheux vous dira que cette suite tend vers 0. Vous pouvez le démontrer en utilisant la formule $e = \sum_{n=0}^{\infty} 1/n!$ et en déduire que $1/n < y_n < 1/(n-1)$.
  Pour comprendre ce que vous observez, il vous faut considérer la suite définie par récurrence par $z_0 = e - 1 + \epsilon$ puis $z_n = n\, z_{n-1} - 1$, et trouver ce que vaut $z_n - y_n$.
  La preuve que $(y_n)_{n\in I\!N} \to 0$ et l'explication du phénomène observé seront fournies plus tard.

- Termes d'une fonction de Syracuse (Partiel L3info 24-25).

  Un #define définit une constante $CSyr$ entier $> 0$. Par exemple `#define CSyr 2025`
  La suite de Syracuse, $(Syr_i)_{i\in N} = Syr_0, Syr_1, Syr_2, ...$, est alors définie par récurrence :
  $Syr_0 = CSyr$. $Syr_n =$ si $Syr_{n-1}$ est pair alors $Syr_{n-1}/2$ sinon $3 * Syr_{n-1} + 1$ Exemples :
  si $CSyr = 7$, la suite est $7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 4, 2, 1, 4, 2, 1, ...$
  si $CSyr = 15$, la suite est $15, 46, 23, 70, 35, 106, 53, 160, 80, 40, 20, 10, 5, 16, 8, 4, 2, 1, ...$

  Donner quatre codes pour la fonction `int Syracuse(int n)` qui rend $Syr_n$
  • Un code itératif
  • Un code récursif terminal avec sous-fonction
  • Un code récursif terminal avec sous-procédure
  • Un code récursif sans sous-fonctionnalité

  Il est interdit de définir la fonction `int delta(int c) {si c pair alors c/2 sinon 3*c+1}`

  Si CSyr=2025, alors $Syracuse(0)$=2025, $Syracuse(3)$=1519, $Syracuse(10)$=15388, $Syracuse(100)$=638, $Syracuse(1000)$=4. Vous devez obtenir ces résultats avec toutes les versions.

  Note : Les mathématiciens conjecturent que cette suite finit toujours par tourner sur 1,4,2. (après 112 itérations pour CSyr=2024 et 156 pour 2025)

- Les permutations P de [0..n-1]=[0..n[ sont représentées par des tableaux P[] d'entiers.
  Rappel, une permutation est une fonction bijective de [0..n-1] dans [0..n-1]. P[i] contiendra l'image de i. Exemple, si P=[4,5,2,1,3,0] alors 4 est l'image de 0, 5 est l'image de 1, etc., et dans $P^{-1}$, ce sera donc 0 l'image de 4, 1 l'image de 5, etc. Codez les fonctionnalités suivantes :

  `P_identite(n)` qui rend la permutation identité. Si n=3, cette fonction rendra [0,1,2]

  `P_inverse(P,n)` qui rend $P^{-1}$. Elle rendra [5,3,2,4,0,1] si n=6 et P=[4,5,2,1,3,0]

  `P_compose(P,Q,out R,n)` qui calcule la composée $PoQ$ des deux permutations P et Q et écrit le résultat dans R, supposé alloué avant l'appel de la procédure. Avec n=6, si P=[0,5,3,4,2,1] et Q=[4,5,2,1,3,0] alors la procédure écrira [2,1,3,5,4,0] dans R (0 donne 4 par Q, et 4 donne 2 par P, donc 0 donne 2 par PoQ)

  `P_verife(P,n)` qui rend vrai ssi le tableau représente bien une permutation de [0..n[, cette fonction rendra faux sur [1,0,3,1], ainsi que sur [0,1,23,42].

  `P_power(P,n,k)` qui rend la puissance k$^{\text{ième}}$ de P. Vous écrirez 4 versions :
  une version récursive qui utilise le fait que $P^{100} = P^{99}oP$, une version itérative qui utilise le fait que $P^{100} = P^{99}oP$, une version récursive plus efficace, une version itérative plus efficace.
  Attention aux fuites mémoire !!! Évitez de désallouer-réallouer à tour de bras.

  `P_random(n)` qui rend une permutation aléatoire. Toutes les permutations doivent pouvoir sortir avec la même probabilité.

# 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 [23,19,42,4,2], [2,-2] et faux sur [2,3,27,1], [2]

- **PlusCourte** qui prend deux listes $L1$ et $L2$ en argument et rend vrai ssi la première est strictement plus courte que la seconde. Si l'une des listes est très courte et l'autre très longue, votre fonction doit répondre rapidement.

  Faire une version récursive **PlusCourteRec** et une version itérative **PlusCourteIter**.

- **Verifiek0** qui prend une liste $l$ et un entier $k$ (supposé positif ou nul) et rend vrai ssi le nombre d'occurences de 0 dans $l$ est $k$. (`Verifiek0([2,0,0,7,0,6,2,4,0],4)` rend vrai, `Verifiek0([2,0,0,7,0,6,2,4,0],1)` rend faux, remarquez que dans ce dernier cas, il n'est pas utile de parcourir la liste jusqu'à la fin pour connaître la réponse).

  Faire une version récursive **Verifiek0Rec** et une version itérative **Verifiek0Iter**.

- **NombreTermesAvantZero** qui prend une liste en argument et rend le nombre de terme avant le premier 0. S'il n'y a pas de 0, on fera comme s'il y en avait un implicite en fin de liste. La fonction rendra 4 sur [3,2,9,5,0,6,0] et sur [3,2,9,5].

  Donnez quatre versions de cette fonction :
  - Une version itérative
  - Une version récursive sans sous-fonctionnalité (et non terminale)
  - Une version qui utilise une sous-fonction récursive terminale avec un compteur "in".
  - Une version qui utilise une sous-procédure récursive terminale avec un compteur "inout".

- **TuePos** procédure qui prend une liste en argument et élimine les éléments qui sont égaux à leur position dans la liste. Exemple, si L=[0,4,3,9,5,0,9,2,1] avant l'appel, alors L=[0,4,9,0,9,2,1] après (le 3 était en 3e position et le 5 en 5e position).

  Faire une version récursive **TuePosRec** et une version itérative **TuePosIt** (Pour cette dernière, ne soyez pas lourd, faites des pointeurs de pointeurs !)

- **TueRetroPos** procédure qui prend une liste en argument et élimine les éléments qui sont égaux à leur position depuis la fin dans la liste. Exemple, si L=[0,4,3,9,5,0,9,2,1] avant l'appel, alors L=[0,4,3,9,0,9] après (le 5 était en 5e retro-position, le 2 en 2e retro-position et le 1 en 1e retro-position). **Ne faire qu'une seule passe**.

## 2.2 Bis

Cette partie ne sera pas lue dans le pré-rendu.

Coder la fonction **PPQ** qui prend en arguments les entiers $p1 > 0$, $p2 > 0$ et $q \geq 0$ et rend la liste de toutes les listes ne contenant que des entiers entre $p1$ et $p2$ et dont la somme vaut $q$. Les listes seront données dans l'ordre alphabétique.

Exemple, $PPQ(2, 4, 9)$ rendra
$[[2, 2, 2, 3], [2, 2, 3, 2], [2, 3, 2, 2], [2, 3, 4], [2, 4, 3], [3, 2, 2, 2], [3, 2, 4], [3, 3, 3], [3, 4, 2], [4, 2, 3], [4, 3, 2]]$

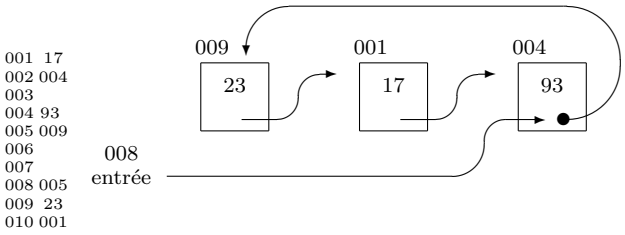
Pour cela, vous vous inspirerez de `Permutations` du cours (cf. le poly.).
L'idée est la suivante : Pour $PPQ(2, 4, 9)$ :
Il vous faut les listes commençant par 2. Si vous avez la liste des listes de nombres entre 2 et 4 dont la somme fait 9-2=7, soit $[[2, 2, 3], [2, 3, 2], [3, 2, 2], [3, 4], [4, 3]]$, alors Vous obtiendrez ce premier jeu de solutions pour q=9 en ajoutant 2 devant chaque liste, ce qui donne $[[2, 2, 2, 3], [2, 2, 3, 2], [2, 3, 2, 2], [2, 3, 4], [2, 4, 3]]$.
De la même manière, en ajoutant 3 devant chaque liste de la liste des listes de nombres entre 2 et 4 dont la somme fait 9-3=6, soit $[[2, 2, 2], [2, 4], [3, 3], [4, 2]]$, vous obtiendrez les listes pour q=9 commençant par 3 $[[3, 2, 2, 2], [3, 2, 4], [3, 3, 3], [3, 4, 2]]$.
Et enfin, vous obtiendrez $[[4, 2, 3], [4, 3, 2]]$ à partir de $[[2, 3], [3, 2]]$
Il ne restera plus qu'à concaténer les trois listes obtenues.

Remarquez que si $q == 0$, alors il y a une liste de termes dont la somme vaut $q == 0$ qui est la liste vide. Puisqu'il faut rendre la liste des listes, vous rendrez $[[]]$
Remarquez que si $0 < q < p1$, alors il n'est pas possible d'obtenir $q$ comme somme de nombres entre $p1$ et $p2$, vous devrez donc rendre $[]$

Le code de `Permutations` du cours occasionne en réalité des fuites mémoires, qui risquent de se retrouver dans votre code. Vous pourrez essayer de les repérer (faites un CptMalloc++ chaque fois que vous faites un malloc pour regarder si vous en obtenez bien autant que prévu) et de les éliminer.
Il est également possible de faire de la compression de mémoire.

## 2.3 Ter

Implémentez les files avec une liste circulaire et un pointeur sur le pointeur dans le dernier bloc (i.e. mettez en place en même temps les deux variantes du bas de la page 24 du poly.) Écrire les fonctionnalités de base dont `entree(in int x, inout file F)` et `sortie(out int x, inout file F)`. Vous manipulez des triples pointeurs ? Oui, c'est normal...



# 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

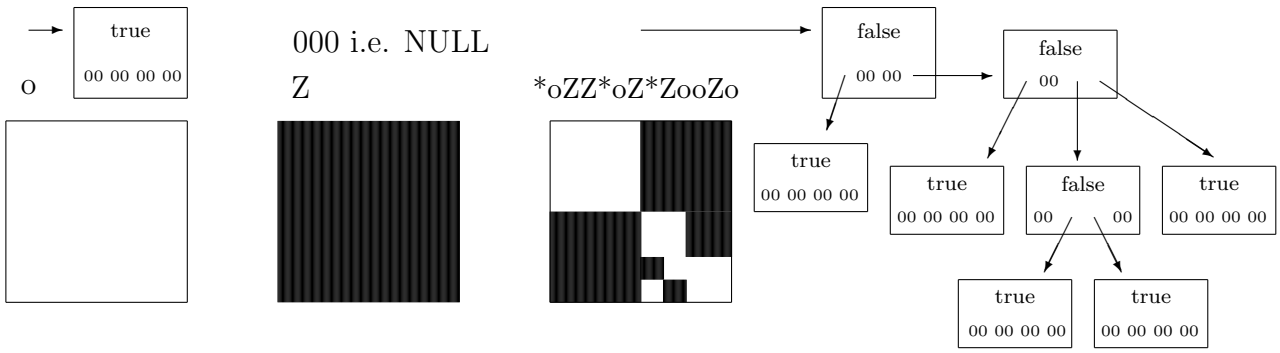
On représentera ces images avec la structure suivante :

```
typedef struct bloc_image
        { bool blanc ;
          struct bloc_image * Im[4] ;
        } bloc_image ;
typedef bloc_image *image ;
```

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

Quand il pointe vers un struct dont le champ `blanc` est `true`, l'image est blanche et les 4 champs `Im[i]` sont NULL.

Quand il pointe vers un struct dont le champ `blanc` est `false`, l'image est obtenue en découpant l'image en 4, et en plaçant respectivement les images `Im[0], Im[1], Im[2], Im[3]` en haut à gauche, en haut à droite, en bas à gauche, en bas à droite.




## 3.1 Entrées Sorties

On utilisera la notation suivante pour les entrées sorties :

- o pour une image blanche
- Z 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 **oooZ*ooZo*oZoo*Zooo 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, *Z*ooZoo*Z*ZZo*ZoZZoZ sera affiché comme suit :

*0 Z1 *1 o2 o2 Z2 o2 o1 *1 Z2 *2 Z3 Z3 o3 *3 Z4 o4 Z4 Z4 o2 Z2

Il est autorisé de rajouter des parenthèses à l'écriture (mais pas de les imposer à la lecture):
*o*ZZoZZ*o*ooZ*oZooZo peut s'afficher (*o(*ZZoZ)Z(*o(*ooZ(*oZoo))Zo))

Lecture : Les caractères autres que oZ* sont sans signification et seront ignorés à la lecture.

## 3.2 Fonctionalités à écrire

Écrire les fonctions et procédures :

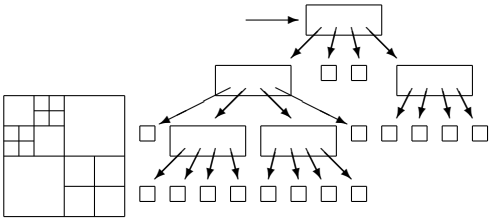
Rappel : le nom des fonctions doit être conservé, nous devons pouvoir retrouver rapidement une fonction par un contrôle F, et nous devons pouvoir injecter votre fonction dans notre programme sans avoir à modifier des noms.

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

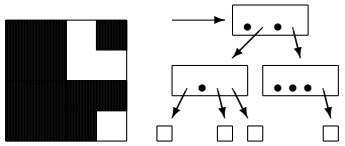
2. Affiche(image) qui affiche son argument en mode simple

3. ProfAffiche(image) qui affiche son argument en mode profondeur

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



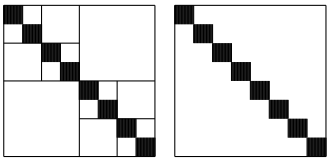
5. DessinNoir(image) et DessinBlanc(image) qui testent si l'image en argument est noire, resp. blanche.
   **o*oooo*ooooooo*oooo est blanche.
   // (*(*o(*oooo)(*oooo)o)oo(*oooo))



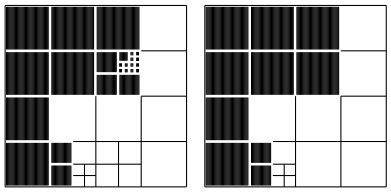
6. QuotaNoir(image) qui rend le taux de noir de l'image argument,
   QuotaNoir(*Z*oZooZ*ZZZo)= 0.75
   // (*Z(*oZoo)Z(*ZZZo))

7. Copie(image) qui rend un nouvel arbre image avec de nouveaux blocs mémoire ayant la même structure que l'arbre image en argument

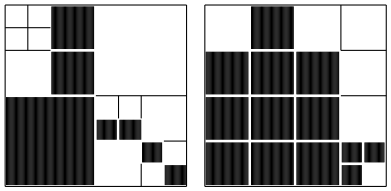
8. Diagonale(int p) qui rend une image qui est noire sur les pixels de profondeur p qui sont sur la diagonale.
   Diagonale(3) rendra ***ZooZoo*ZooZoo**ZooZoo*ZooZ
   (*(*(*ZooZ)oo(*ZooZ))oo(*(*ZooZ)oo(*ZooZ)))



9. SimplifieProfP qui prend en argument un arbre et une profondeur p et le TRANSFORME en remplaçant tous les arbres monochromes à profondeur p par de simples pixels blancs ou noirs. Exemple, p=2, l'arbre * (*ZZZZ) (*Zo(*Z(*Z(*ZZZZ)(*ZZZZ)(*ZZZZ))ZZ)o)
   (*ZoZ(*ZoZ(*oooo))) (*oo(*oooo)o)
   est transformé en *(*ZZZZ)(*ZoZo)(*ZoZ(*ZoZ(*oooo)))(*oooo)



10. Incluse qui prend deux images en argument, et qui rend vrai ssi la première est incluse dans la seconde, i.e. que la seconde est noire partout où la première est noire.
    ***ooooZoZoZ**ooZZoo*ZooZ n'est pas incluse dans
    **oZZZ*ooZo*ZZZZ*ZoZ*ZZZo à cause du pixel tout en bas à droite.

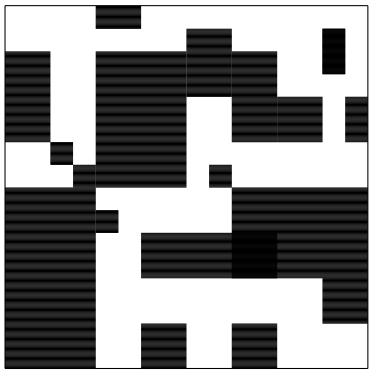


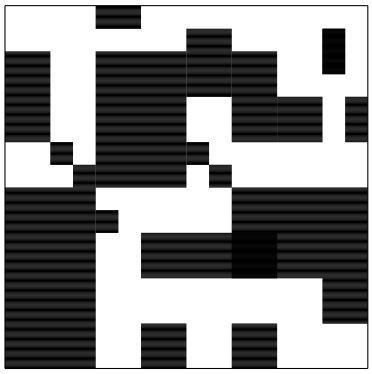
11. CompteSousImagesGrises(image). Une image grise est une image dont le quota noir est compris entre 1/3 et 2/3. La fonction rendra le nombre de sous images grises. L'image *Z*oZooo*Zooo est grise. L'image *oZ*Z*oZooo*Zooo *Z*oZooo*ZZoo contient 4 sous-images grises (elle-même, ses sous images bas-gauche et bas-droite, et la sous image bas-droite de sa sous-image bas-droite)

12. Le dessin représente un labyrinthe. Les zones noires sont des murs infranchissables, vous partez du coin en haut à gauche et souhaitez arriver au coin en bas à droite en circulant dans les pixels blancs, vous pouvez passer d'un pixel à un autre par les côtés mais pas par les angles.

    La fonction Labyrinthe rend un booléen qui dit si la traversée est possible. Elle rendra

    vrai sur ***ooZo**ZZoooZZ*Zoo*ZooZZ ***ooZZoZZ*o*ooZoo*Zooo*oZ*oooZo*Z*oZoZoo *Z**ooZoooZZ*oooZ **oZZZZ*oooZ*oZoo
    faux sur ***ooZo**ZZoooZZ*Zoo*ZooZZ ***ooZZoZZ*o*ooZoo*Zooo*oZ*ZooZo*Z*oZoZoo *Z**ooZoooZZ*oooZ **oZZZZ*oooZ*oZoo





# 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

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

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 $\epsilon$ 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 toute 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 $Y_{400000}$.