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.

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...

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 Interclassements et ListesZ
- Dupont2IC.c pour Interclassements
- Dupont2Z.c pour les ListesZ
- 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)

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, Interclassements 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 17 octobre matin
- prérendu de la partie 2 le 19 octobre matin
- rendu définitif des parties 1 et 2 le 2 décembre à 8h
- prérendu de la partie 3 le 11 décembre matin
- rendu définitif de la partie 3 le 13 janvier à 8h

Deux fichiers Algo1.c et Algo2.c sont disponibles sur ecampus, 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.

  Faire une version qui rend un float et une version qui rend un double.

- 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 et une version avec un double. Que constatez-vous ?

  Vous souhaitez l'explication ? Considérez la suite définie par récurrence par $z_0 = e - 1 + \epsilon$ et $z_n = n\, z_{n-1} - 1$, et trouvez ce que vaut $z_n - y_n$.

  La preuve que la suite tend vers 0 et l'explication du phénomène informatique vous seront fournies plus tard.

- La suite de réels $(x_n)_{n\in N}$ est définie par récurrence: $x_0 = 1$ puis $\forall n \geq 1, x_n = x_{n-1} + 1/x_{n-1}$.

  On a donc $x_0 = 1$, $x_1 = 1 + 1/1 = 2$, $x_2 = 2 + 1/2 = 2.5$, $x_3 = 2.5 + 1/2.5 = 2.9$, etc.

  Coder la fonction $X(n)$. Donner quatre versions :
  - X1, une version itérative,
  - X2, une version récursive sans utiliser de sous-fonctionnalité,
  - X3, une version récursive terminale avec sous fonction
  - X4, une version récursive terminale avec sous procédure

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

  Tentez de calculer X_{10^k} pour k de 1 à 9 avec chacune des versions.
  Observez ce qui passe avec $X2$.
  $X3$, $X4$: Votre compilateur semble-t-il optimiser le récursif terminal ?

  Les résultats pour X_1 semblent-ils corrects ?
  Recodez $X1$ avec n en `long` au lieu de `int` et type de sortie `long double` au lieu `float` , puis tentez de calculer X_{10^k} pour k de 1 à 12. Observez.

- Les nombres binomiaux (Exam CFA 23-24)

  Vous voulez écrire une fonction qui calcule les nombres binomiaux $Cpn(p,n) = C_n^p = \binom{n}{p}$.

  Vous n'avez que de très lointains souvenirs de math. Il y avait une formule avec des factorielles, mais pas moyen de la retrouver (en clair, interdit de l'utiliser pour les codes demandés).

  Par contre, vous vous souvenez d'une formule par récurrence ("triangle de Pascal") qui disait:

  $\forall\, n, C_n^0 = C_n^n = 1$

  $\forall\, 0 < p < n, C_n^p = C_{n-1}^p + C_{n-1}^{p-1}$

  - Écrivez un premier code récursif directement inspiré de la formule de récurrence
  - Tentez d'afficher les valeurs de C_{2*n}^n pour n de 0 à 30
  - Proposez un 2e code.

  Le type de sortie sera "`long`" ("`int`" n'a pas assez d'octets pour gérer C_{60}^{30}).

2 Listes-Piles

2.1

- **DeuxEgalX** qui prend en argument une liste L d'entiers et un entier x et rend vrai ssi le deuxième élément de L est égal à x. Si L n'a pas de deuxième élément, le deuxième élément sera supposé être 0. Exemples, la fonction rend vrai sur ([3,5,8,4,29],5) et sur ([2],0) et faux sur ([3,5,8,4,29],6) et sur ([2],1).

- **ContientZero** qui prend une liste en argument et rend vrai ssi il y au moins une occurence de 0 dans la liste. Faire une version récursive et une version itérative.

- **SousEnsemble** qui prend en entrée deux listes l_1 et l_2 supposées triées dans l'ordre croissant, et sans doublons, et rend vrai ssi $l1$ est un sous-ensemble de $l2$.

- **SommeAvantKieme0** qui prend en argument une liste d'entiers L et un entier positif ou nul k et rend la somme des éléments positionnés avant le k^e 0. S'il y a moins de k 0, la fonction rend la somme des éléments. Exemple, SAvK0([2,3,0,1,0,4,9,0,**0**,8,2,0],4) rend $2+3+1+4+9=19$. Écrire 4 versions :
  • Une version récursive (non terminale) sans sous-fonctionnalité.
  • une version itérative.
  • une version avec sous-fonction f(arguments in) récursive terminale
  • une version avec sous-procédure void p(in L,k, inout ...) récursive terminale

- **SommeApresRetroKieme0** qui prend en argument une liste d'entiers L et un entier k et rend la somme des éléments positionnés après le rétro-k^e 0. S'il y a moins de k 0, la fonction rend la somme des éléments. Exemple, SApRK0([2,3,0,1,**0**,4,9,0,0,8,2,0],4) rend $4+9+8+2=23$. Ne faire qu'une seule passe.

- **TueDoublons** procédure qui prend une liste en arguments et élimine des éléments pour ne garder qu'une occurrence de chaque élément présent dans la liste. Faire deux versions. Dans la première version, seule la dernière occurrence est conservée. Dans la seconde version, seule la première occurrence est conservée. Exemple, si $L == [34, 56, 34, 23, 12, 34, 23]$. Après l'appel de `TueDoublons1(L)`, $L == [56, 12, 34, 23]$. Après l'appel de `TueDoublons2(L)`, $L == [34, 56, 23, 12]$.
  Faire une version récursive pour la version1.
  Faire une version récursive et une version itérative pour la version2.
  La complexité attendue est quadratique. Faire des versions quadratiques. (Pour information, il est possible et intéressant de se demander comment faire mieux.)
  Ne pas utiliser `miroir`.

2.2

Cette partie ne sera pas lue dans le prérendu.

- Coder la fonction **Interclassements** en utilisant la technique "diviser pour régner" du cours (confer le TD2 et le poly).

  L'espace mémoire du résultat devra être indépendant de l'espace mémoire des arguments. Ajoutez des compteurs pour compter le nombre de malloc effectués (un pour chaque type). Observez la fuite mémoire sur l'un des types. (Délicat :) Parvenez-vous à éliminer la fuite mémoire ?

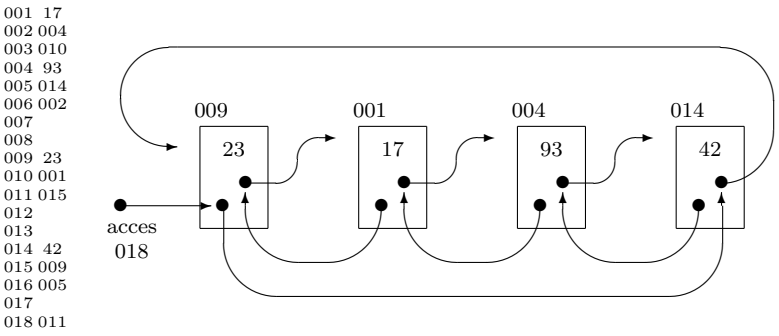
- Les ListesZ sont construites avec des blocs. Chaque bloc possède trois champs, un champs `valeur` de type entier, un champs `next` qui pointe vers le bloc suivant, et un champs `prec` qui pointe vers le champs `next` du bloc précédent. S'il n'y a qu'un seul bloc, les blocs suivant et précédent sont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

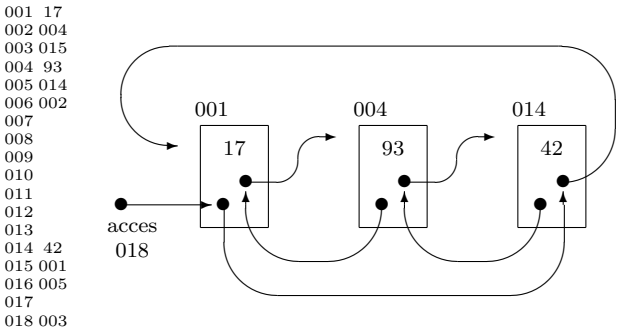
  Écrire le code de la procédure `ZElimine` qui prend en argument un pointeur `acces` et enlève le bloc à l'intérieur duquel `acces` pointe, qui sera rendu à la mémoire.

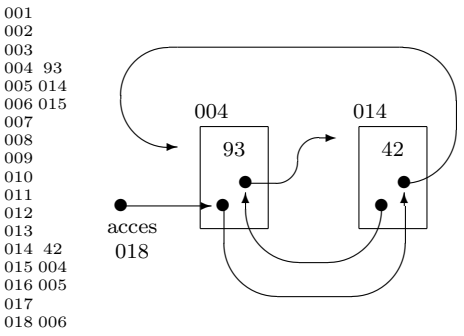
  Après l'appel, `acces` à l'intérieur du bloc qui suivait celui qui a été éliminé, sous réserve que celui-ci existe.

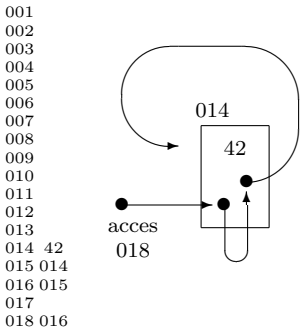
  Si la liste est vide, `Zelimine` ne fait rien (elle ne plante pas)

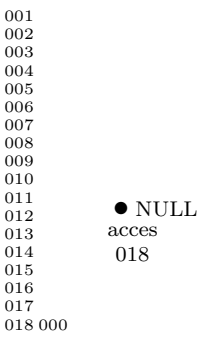
  Exemple, si la liste contient 23, 17, 93, 42 selon le premier schéma ci-dessous, et que l'on effectue plusieurs fois `Zelimine`, on obtient successivement les listes ci-dessous.











2.3 Quelques remarques

Aérez bien et mettez des /******/ souvent pour faciliter la correction. Tiltez que j'aurai 130 codes à lire. Je m'autorise à enlever des points aux codes pénibles.

Vos tests ne sont pas demandés et je ne les regarderai a priori pas. Si vous les laissez, mettez-les après l'ensemble des fonctions (par exemple dans le main).
Par contre, je serai sévère si un bug reste qui aurait dû être repéré par des tests.

Je vous invite à regarder quand il y a une instruction du genre "regardez ce qui se passe", mais il n'est pas demandé de rapport. Parfois, c'est juste pour votre apprentissage. Parfois, c'est pour que vous détectiez un éventuel problème de votre code.

N'utilisez pas de **Undefined Behavior**, ni de **comportements non standards**. Sinon votre code compilera et fonctionnera chez vous mais pas chez vos copains et 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.

Certains compilateurs mettent 0 par défaut dans les variables déclarées, mais d'autres laissent ce qui y traîne. Plusieurs codes des prérendus, impliquant des tableaux, 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

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.

Fonction X :

Itératif : Il est anormal que vous commenciez par tester si n vaut 0, soit cela ne sert à rien, soit cela pallie un mauvais codage de la boucle.

Récursif simple : Si votre code ne permet pas de calculer X_{100}. vous avez un problème... Voir par exemple TD1 Exo4 RE2 et RE3

Récursif terminal : N'oubliez pas les surfonctions. Mes sous-fonctionnalités n'ont que deux arguments.

Si besoin, révisez la correction de power dans le TD1

Good Cpn

Quelques petites optimisations sont possibles.

Une grosse optimisation en mémoire est possible.

Rappel : L'énoncé impose d'utiliser le triangle de Pascal. Il vous est donc interdit d'utiliser factorielle.

DeuxEgalX :

Beaucoup de codes ne rendent pas toujours la bonne réponse sur liste vide ou singleton (la réponse dépend de x), ou font un Segmentation Fault.

L'appel à `longueur` est catastrophique pour la complexité.

À part dans DeuxEgalX, si vous avez un test `L->suite == NULL`, c'est a priori une lourdeur.

Sous-ensemble :

[2,4,5,6,7,8] n'est pas sous-ensemble de [3,4,5,6,8,9] et il n'est pas nécessaire de parcourir les listes jusqu'au bout pour s'en apercevoir, alors ne parcourez pas les listes jusqu'au bout.

Il serait bien que votre fonction se généralise aux multi-ensembles. Dans un multi-ensemble, un élément peut apparaître plusieurs fois. Et pour que l_1 soit sous-ensemble de l_2, il faut que chaque élément apparaisse au moins autant de fois dans l_2 que dans l_1. Exemple, [2,2] est sous-ensemble de [1,2,2,2,3] et de [1,2,2,3] mais pas de [1,2,3]. Si votre fonction ne fait pas cela, n'allez pas faire des choses compliquées, normalement, une seule ligne de code en plus et le tour est joué.

SommeAvantKieme

Certains codes bugent sur si k vaut 0.

SommeApresKieme

Je rappelle que l'énoncé dit une seule passe.

Tue-doublons

Essayez de faire simple. J'ai codé TD1 en récursif (terminal) et en itératif. TD2 itou. Je compte une ligne par en-tête, une ligne par test, une ligne par instruction. Mes 4 codes font 40 lignes en tout, de 9 à 12 lignes par version.

Utiliser un pointeur vers le dernier bloc de la liste est moyen. Ce n'est pas comme cela que vous aurez un code cours. Essayez d'utiliser un pointeur de pointeur.

Dans la version récursive, tout doit être récursif, la sous-fonction et la sur-fonction. Dans la version itérative, tout doit être itératif.

J'ai vu une version bugée chez un binôme qui n'avait pas fait de test qui montre le bug. Je vous suggère de tester sur [1,2,1,2].

Ici comme ailleurs, si vous avez un test `L->suite == NULL`, c'est a priori une lourdeur. Vous économisez peut-être une itération ou un appel, mais vous rajoutez ou compliquez des tests.

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

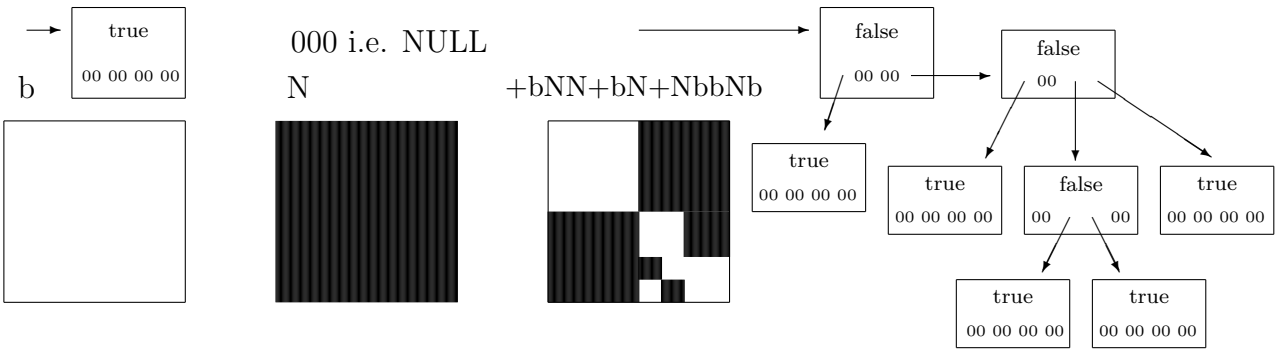
On représentera ces images avec la structure suivante :

```
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 :

- b pour une image blanche
- N 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 ++bbbN+bbNb+bNbb+Nbbb 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, +N+bbNbb+N+NNb+NbNNbN sera affiché comme suit :

  +0 N1 +1 b2 b2 N2 b2 b1 +1 N2 +2 N3 N3 b3 +3 N4 b4 N4 N4 b2 N2

Il est autorisé de rajouter des parenthèses à l'écriture (mais pas de les imposer à la lecture):
+b+NNbNN+b+bbN+bNbbNb peut s'afficher (+b(+NNbN)N(+b(+bbN(+bNbb))Nb))

• En mode 2^k-pixel, l'affichage se fait sur 2^k lignes et 2^k colonnes, en utilisant le point pour le blanc, le 8 pour le noir, et le − quand la résolution de l'affichage est insuffisante pour donner une couleur.
L'affichage 2^3-pixel de +N+bbNbb+N+NNb+NbNNbN donne :

```
8888....
8888....
888888..
888888..
....8888
....88.-
......88
......88
```

Lecture : Les caractères autres que bN+ 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.

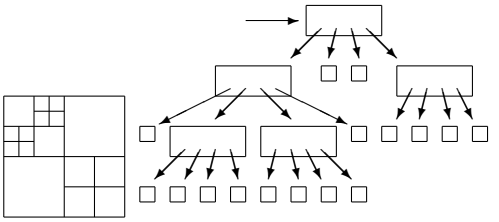
1. Bc() qui rend une image blanche à partir de rien.
   Nr() qui rend une image noire à partir de rien.
   Qt(i0,i1,i2,i3) qui construit une image composée des sous-images i0,...,i3. (`Qt` pour `Quatre`)

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

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

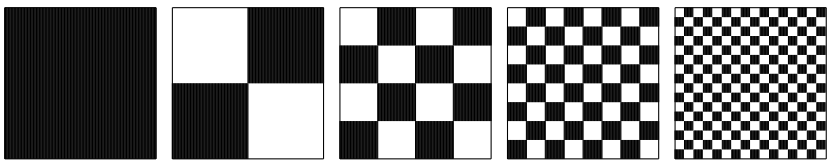
4. LireI() qui rend une image à partir des caractères tapés au clavier.

   Il est suggéré d'utiliser getchar() qui lit un caractère.



5. Noir(image) et Blanc(image) qui testent si
   l'image en argument est noire, resp. blanche.
   ++b+bbbb+bbbbbbb+bbbb est blanche.
   // (+(+b(+bbbb)(+bbbb)b)bb(+bbbb))

6. Damier(p) qui rend un damier
   de profondeur p.
   Ci-contre les damiers de
   profondeur 0,1,2,3 et 4



7. DemiTour(image) qui rend l'image en argument tournée d'un demi-tour. DemiTour de ++bNNb+bbbN+NNbbN rend +N+bbNN+Nbbb+bNNb

8. FreeImage(image) qui rend tous les blocs d'une image à la mémoire

9. Simplifie qui prend en argument un arbre et le TRANSFORME en remplaçant tous les arbres monochromes par de simples pixels blanc ou noir. Exemple, l'arbre

   +(+NNNN)(+Nb(+NN(+N(+N(+NNNN)NN)NN)N)b)(+NbN(+NbN(+bbbb)))(+bb(+bbbb)b)

   est transformé en +N(+NbNb)(+NbN(+NbNb))b

   Il est possible de faire une procédure linéaire. C'est à la fois simple et un peu subtil.

10. IntersectionVide qui prend deux images en argument et rend vrai ssi leur intersection est blanche. Si les images sont ++bNbN+NbbNN+bbNN et +b+bNbbb+bNbN , la fonction rendra faux car l'intersection n'est pas blanche sur le pixel en bas à droite.

    Cherchez à gérer habilement les cas de base.

11. CompteSousArbres qui prend deux images I1 et I2 et dit combien de fois I1 est sous-arbre de I2, c.-à-d. combien de fois I1 est identique au sous-arbre enraciné en x nœud de de I2. Exemples :

    +b+bNNbNb est sous arbre une fois de +b+bNNbNb et deux fois de
    +N(+b(+bNNb)Nb)(+b(+b(+bNNb)Nb)Nb)N

    +b(+NNNN)Nb n'est pas sous arbre de +N(+bNNb)bN, et +bNNb n'est pas sous arbre de +N(+b(+NNNN)Nb)bN. (Ils sont sous-images mais pas sous-arbres)

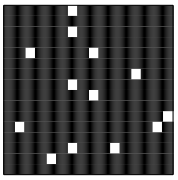
    Note : il pourra être intéressant, pour une meilleure complexité (linéraire...), de ne regarder si le sous-arbre enraciné en x est identique à I1, que si la hauteur de x est celle de I1.

    Une meilleure note sera attribuée si le résultat se calcule via une variable `cpt` "inout".

12. PrintPix(image,k) qui affiche son premier argument en mode 2^k-pixel. Il est largement préférable de ne pas utiliser un tableau de taille variable (4^k par exemple). Il est interdit d'utiliser tout autre utilitaire évolué (bibliothèque d'affichage graphique) pour cet affichage. In fine, n'utilisez que printf("."); printf("8"); printf("-"); printf("\n");

13. Alea qui prend en argument une profondeur k, et un entier n et qui rendra une image dont la partie blanche sera constistuée de n pixels blancs à profondeur k, positionnés aléatoirement. Chaque image pouvant sortir de préférence équiprobablement.
    Par exemple, Alea(1,2) rendra chacune des images
    +NNbb, +NbNb, +NbbN, +bNNb, +bNbN, +bbNN avec probabilité 1/6

    tandis que Alea(4,13) pourrait rendre l'image ci-contre :

    Suggestion de quelques tests :
    Alea(0,0)
    Alea(0,1)
    Alea(100,1)
    Alea(10,1000) (Faire un affichage pixel)
    Alea(2,2) un milliard de fois.
    Vérifiez que ++NNbbbbb sort à peu près aussi souvent que ++Nbbbbb+bbbN

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 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 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}.