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 fonctions en C, qui font manipuler explicitement récursivité, pointeurs, listes chaînées, passages par adresse, etc. et qui poseront quelques problèmes algorithmiques. Attention, C est très permissif voire pousse-au-crime. On peut facilement y programmer très salement. C'est à vous d'être propre :

- faire du code lisible, bien structurer les programmes, bien les présenter. Un bon code est compréhensible par quelqu'un qui ne connaît pas le langage dans lequel il est écrit.
- 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;
```

Notation : Partiels et exams d'algo sont un peu 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. Le principe de notation est le suivant : Une note brute est donnée, puis les points bruts donnent des points sur 20 de façon dégressive. Ordre d'idée : note brute sur 170, puis 1 point sur 20 pour 5 points bruts jusqu'à 10 (donc 50/170 donne 10), puis pour 10 points jusqu'à 18 (donc 130/170 donne 18) puis pour 20 points (donc 160/170 donne 19.5).

Travailler le projet donne donc des points d'avance pour l'U.E., mais il permet aussi de s'entraîner pour les partiel et examen et aide à y avoir une note correcte.

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), ou pire, qui ne compile pas.

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.

Dates de rendus. Pour les rendus (1) et (3), nous tenterons d'envoyer les commentaires avant le partiel et l'examen. Les rendus en retard seront corrigés et commentés en dernier.

(1) 20 octobre, parties 1 et 2. Ce rendu sera corrigé, commenté mais non noté.

(2) 21 novembre 9h, parties 1 et 2. Ce rendu sera noté.

(3) 5 décembre 9h, partie 3. Ce rendu sera corrigé, commenté mais non noté.

(4) 9 janvier 9h, partie 3. Ce rendu sera noté

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 pouvez pas sommer pas jusqu'à l'infini... Il est pertinent de vous demander quand et comment vous vous arrêtez.

- On définit la suite $y_0 = e - 1$, puis par récurrence $y_n = n\, y_{n-1} - 1$. Faire afficher les 50 premiers termes. Que constatez-vous ? Augmentez la précision de la valeur de départ en utilisant un double. Observez. Avez-vous une explication ?

- Implémentez les 10 versions de power qui sont numérotées dans la correction du TD1 (confer icelle sur ecampus)

  Pour les deux premières (récursive et itérative naïves), écrire des fonctions qui fonctionnent avec n positif ou négatif (Note 0^{-1} plante, c'est normal...)

  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 10 versions avec N de grosses puissances de 10, $N = 10^9$, $N = 10^{12}$ (au besoin, utilisez des long et des double) et observez : le calcul termine immédiatement, il rame, il tourne sans donner de résultat, il stoppe avec un out of memory ?

  Votre compilateur semble-t-il effectuer l'optimisation du récursif terminal ?

- Implémentez plusieurs méthodes pour calculer la fonction d'Ackermann : Une version purement récursive et une version récurso-itérative avec un pour sur n ont été vues en TD. Il est également possible de faire une version récurso-itérative avec un pour sur m.

  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 de réels $(x_n)_{n\in N}$ est définie par récurrence: $x_0 = 1$ puis $\forall n \geq 1, x_n = x_{n-1} + 2/x_{n-1}$.

  On a donc $x_0 = 1$, $x_1 = 3$, $x_2 = 11/3 = 3.666...$, $x_3 = 139/33 = 4.2121...$

  Coder la fonction $X(n)$. Donner quatre versions : une version itérative, une version récursive sans utiliser de sous-fonctionnalité, deux versions récursives terminales, l'une avec sous-fonction et l'autre avec sous-procédure.

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

2 Listes-Piles

• • Un mini-programme avec en-tête, déclarations et quelques fonctionnalités est founi en annexe. Vous êtes invités à le compléter, en implémentant les fonctions et procédures suivantes :

- **ZeroEnPositionUnOuDeuxOuTrois** qui prend en argument une liste et rend vrai ssi il y a un zéro en première, deuxième ou troisième position. Elle rendra par exemple vrai sur la liste [3,4,0,2,8] ainsi que sur la liste [0,2]. Et faux sur [3,4,5,0,2,8] ainsi que sur la liste [9,2],

- **Pluscourte** qui prend deux listes $L1$ et $L2$ en argument et rend vrai ssi la première est strictement plus courte que la seconde.

- **NombreDe0AvantPositionK** qui prend en argument une liste L et un nombre K et rend le nombre de zéros dans la liste avant ou à la Kième position. Par exemple, si $L = [2, 0, 5, 6, 9, 0, 0, 0, 1, 0]$ et $K = 6$, la fonction rendra 2. Itou pour $L = [5, 0, 0, 3]$. 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 avec argument(s) supplémentaire(s). • Une version qui utilise une sous-procédure récursive terminale avec argument(s) supplémentaire(s).

- **NombreDe0ApresRetroPositionK** qui prend en argument une liste L et un nombre K et rend le nombre de zéros dans la liste à droite de ou à la Kième position depuis le fond. Par exemple, si $L = [0, 1, 0, 0, 0, 9, 6, 5, 0, 2]$ et $K = 6$, la fonction rendra 2. Itou si $L = [5, 0, 0, 3]$ et $K = 6$. Ne faire qu'une seule passe.

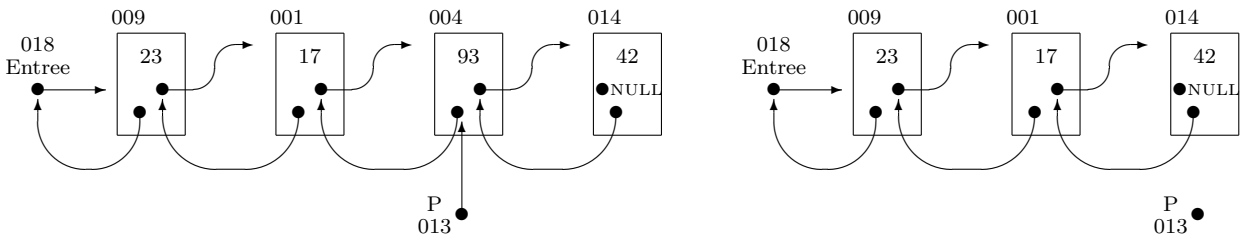
- **FctBegaye** qui ne modifie pas la liste, et rend une liste nouvelle (avec de nouveaux blocs) dans laquelle tous les éléments strictement positifs sont recopiés deux fois, et les autres ne sont pas recopiés. Par exemple, FctBegaye de $[2, 1, 0, 6, -2, 8, 8]$ rendra $[2, 2, 1, 1, 6, 6, 8, 8, 8, 8]$. Faire une fonction récursive simple sans sous-fonctionnalité. Puis (plus délicat) faire une fonction récursive terminale et une itérative. Ne pas utiliser la procédure `ProcBegaye`

- **ProcBegaye** qui modifie la liste en entrée en dédoublant tous les éléments strictement positifs de la liste et en éliminant tous les autres. Exemple, si avant l'appel $l = [2, 1, 0, 6, -2, 8, 8]$ alors l'appel transforme l en $[2, 2, 1, 1, 6, 6, 8, 8, 8, 8]$. Faire du récursif terminal. Ne pas utiliser la fonction `FctBegaye`.

- 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 ?

- Implémentez les ListesBis comme les listes mais avec un champ `pred` qui pointe sur le champ `suivant` du bloc précédans, sauf pour le premier bloc dans lequel `pred` pointe sur le pointeur `Entree` de début de liste. Un pointeur P pointe sur un champs `pred`. Écrire la procédure qui retire de la liste le bloc contenant le champs `pred` pointé par P et le rend à la mémoire.

```
001 17
002 004
003 010
004  93
005 014
006 002
007
008
009  23
010 001
011 018
012
013 006
014  42
015 000
016 005
017
018 009
019
020
```



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

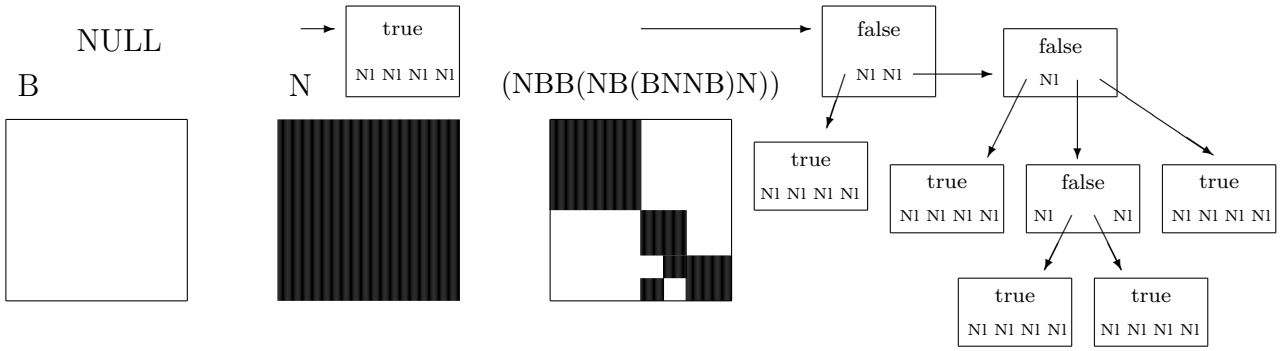
On représentera ces images avec la structure suivante :

```
typedef struct bloc_image
        { bool toutnoir ;
          struct bloc_image * fils[4] ;
        } bloc_image ;
typedef bloc_image *image ;
```

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

Quand il pointe vers un struct dont le champ `toutnoir` est `true`, l'image est noire et les 4 champs `fils[0], fils[1], fils[2], fils[3]` sont NULL.

Quand il pointe vers un struct dont le champ `toutnoir` est `false`, l'image est obtenue en découpant l'image en 4, et en plaçant respectivement les images `fils[0], fils[1], fils[2], fils[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'image ((BBBN)(BBNB)(BNBB)(NBBB)) est un carré noir au centre de l'image.

Les caractères autres que () B N sont sans signification et doivent donc être ignorés à la lecture.

- Le mode simple affiche une image selon le mode ci-dessus.
- En mode profondeur, le degré de profondeur est donné après chaque symbole. Par exemple, (N(BBNB)B(N(NNB(NBNN))BN)) sera affiché comme :

(N1 (B2 B2 N2 B2) B1 (N2 (N3 N3 B3 (N4 B4 N4 N4)) B2 N2))

3.2 Fonctionalités à écrire

Écrire les fonctions et procédures :

- Construit_Blanc() qui rend une image blanche à partir de rien.
  Construit_Noir() qui rend une image noire à partir de rien.
  Construit_Composee(i0,i1,i2,i3) qui construit une image composée des sous-images i0,...,i3.

- Affichages en modes normal et profondeur.

- Lecture qui rend une image à partir des caractères tapés au clavier

- EstNoire et EstBlanche qui testent si l'image en argument est noire, resp. blanche.
  (BBB(BB(BBBB)B)) est blanche.

- Diagonale(int p) qui rend une image qui est noire sur les pixels de profondeur p qui sont sur la diagonale. Diagonale(3) rendra (((NBBN)BB(NBBN))BB((NBBN)BB(NBBN)))

- QuartDeTour qui rend l'image en argument tournée d'un quart de tour dans le sens des aiguilles d'une montre. QuartDeTour de ((BNNN)B(NNBB)N) rend ((BNBN)(NBNN)NB)

- Negatif, procédure qui TRANSFORME l'image argument en son négatif

- SimplifieProfP qui prend en argument un arbre et une profondeur P et remplace tous les arbres monochromes à profondeur P par de simples pixels B ou N.
  Exemple, si P=2, l'arbre (N (NB(NN(NNNN)N)B) (NBN(NBN(BBBB))) (BB(BBBB)B))
  est transformé en (N (NBNB) (NBN(NBN(BBBB))) (BBBB))

- 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.
  (((BBBB)NBN)BN((BBNN)BB(NBBN))) n'est pas incluse dans
  ((BNNN)(BBNB)(NNNN)(NBN(NNNB))), à cause du pixel tout en bas à droite.

- HautMaxBlanc qui prend une image et rend le maximum des hauteurs des sous-images blanches. HMB de ((BBBB) N (BNBN) (NBN(NB(BB(BBBB)(BBB(BBBB)))N))) rendra 3 : les B sont de hauteur 0, les (BBBB) de hauteur 1, (BBB(BBBB)) est de hauteur 2 et (BB(BBBB)(BBB(BBBB))) est de hauteur 3. Si l'image est noire, la fonction rendra -1

- BlanchitProfP qui prend en argument un arbre, une profondeur P, deux entiers x et y compris entre 0 et $2^p - 1$, et blanchit le pixel de profondeur p, de coordonnées (x,y). Le pixel (0,0) est celui en bas à gauche, le pixel ($2^p - 1$,0) est celui en bas à droite. Exemple si p=2, x=1 et y=3, la procédure transforme (NBNB) et ((N(NBNN)NN)BNB) en ((NBNN)BNB).

- Chute. La gravité agit sur les pixels qui tombent et s'empilent dans le bas de l'image. Chute calcule le résultat.
  Exemple Chute (N ((NBNN)NB(BNBN)) (N(NBBN)(BNNN)(NBNB)) (NN(NBBB)(BNBN)))
  rend (((BNNN)(BBNB)N(NBNN)) (B(BNBN)(BBNB)(BNBN)) N ((NBNN)NNN))
  Le NNN final pourra apparaître comme (NNNN)NN