TP Algo en autonomie, LDD2 & L3 UPSaclay

Ce TP est à faire en binôme (ou en monôme). La discussion entre étudiants est autorisée. Le plagiat ne l'est pas. Les deux membres du binôme doivent pouvoir expliquer les codes fournis.

Le but de ce TP est de vous faire programmer quelques petites fonctions en C, qui vous feront manipuler explicitement récursivité, pointeurs, listes chaînées, pasages par adresse, etc. et poseront quelques petits problèmes algorithmiques.

Attention, ce langage 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 connait 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;
```

1 Quelques calculs simples

- Calculez e en utilisant l'expression $e = \sum_{n=0}^{\infty} 1/n!$.

  Evidemment, vous ne sommerez pas jusqu'à l'infini...

- Implémentez la fonction $Puissance(x, n)$, x réél, n entier positif ou nul.

  Utilisez-la pour calculer 1.1^{10}, 1.01^{100}, 1.001^{1000}... $(1 + 10^{-k})^{10^k}$...

  Obersez les différences entre les diverses méthodes de calcul. Y a-t-il un effet visible de précision à utiliser un double plutôt qu'un float ? Sachez que $(1 + \epsilon)^{1/\epsilon}$ vaut environ

  $e * (1 - (1/2) * x + (11/24) * x^2 - (7/16) * x^3 + (2447/5760) * x^4 - (959/2304) * x^5 + O(x^6))$

- Implémentez les deux méthodes pour calculer la fonction d'Ackermann.

- 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 $x_0 = 1$, $x_1 = 2$, $x_2 = 2.5$, $x_3 = 2.9$, etc.

  Ecrire le pseudo-code de la fonction X qui prend n en argument et rend x_n.

  Donner une version itérative et une version récursive (sans utiliser de sous-fonctionnalité).

  Utilisez les deux méthodes pour calculer X_{100}

2 Listes-Piles et Files

• • 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 (issues pour la plupart des partiels de l'an dernier) :

- **DebutDeuxIdentiques** qui prend en argument une liste et rend vrai si et seulement si elle commence par deux éléments identiques. Elle rendra vrai sur $[4, 4]$ et sur $[2, 2, 0, 6, 0]$ mais pas sur $[1, 2, 3]$ ni sur $[]$.

- **QueDesZeros** qui prend une liste en argument et rend vrai ssi tout élément apparaissant dans cette liste est un 0 (en particulier `QueDesZeros(liste vide)` rend vrai).

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

- La fonction **Permutations** vue en cours.

- **EliminePositionsPaires** qui prend en argument une liste L et élimine tous les éléments en position paire. Par exemple, si avant l'appel $L = [2, 1, 6, 8, 8, 3]$ alors l'appel transforme L en $[2, 6, 8]$

- **Begaye** qui modifie la liste en entrée en dédoublant tous les éléments strictement positifs de la liste et en éliminant tous les autres. Par 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.

- **MaxZerosConsecutifs** qui prend une liste en argument et rend le plus grand nombre de zeros consécutifs de la liste. Exemples : MaxZerosConsecutifs([4,8,2,9]) rend 0, MaxZerosConsecutifs([9,7,0,0,0,8,0,7,0,0,0]) rend 3.

  • Faire une version itérative

  • Faire une version récursive avec une sous fonction qui a trois arguments `in` (Reprendre le principe de la version itérative)

  • Faire une version récursive avec une sous fonction qui a un argument `in` et deux arguments `out` (Comme la version précédente, mais on compte de droite à gauche)

- La fonction **EstPalindrome** vue en TD.

- **SommeAvantApres** qui prend une liste d'entiers en argument et rend vrai ssi il y a un élément tel que la somme de tous les éléments qui le précèdent est égal à la somme de tous les éléments qui le suivent. La fonction rendra vrai pour $[2, 3, 0, 4, -2, 7]$ (cf le 4 car $2 + 3 + 0 = -2 + 7$). Ne faire qu'une seule passe

• • 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 20 du poly.) Ecrire les fonctionnalités de base dont `ajoute(in int x, inout file F)` et `sortir(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 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 sans signification. Il est conseillé mais non obligatoire de les mettre a NULL.

Quand il pointe vers un record 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 préfixe pour les entrée sortie. La notation préfixe consiste à écrire

- 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. Ils peuvent servir (notamment le blanc, le retour-ligne les parenthèses) à améliorer la lisibilité des affichages.

- Le mode simple affiche une image en écriture préfixe.

- En mode profondeur, le degré de profondeur est donné après chaque symbole. Par exemple, . N .BBNB B .N.NNB.NBNNBN sera affiché comme :
.0 N1 .1 B2 B2 N2 B2 B1 .1 N2 .2 N3 N3 B3 .3 N4 B4 N4 N4 B2 N2

- 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 .BBNB B .N.NNB.NBNNBN donne :

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

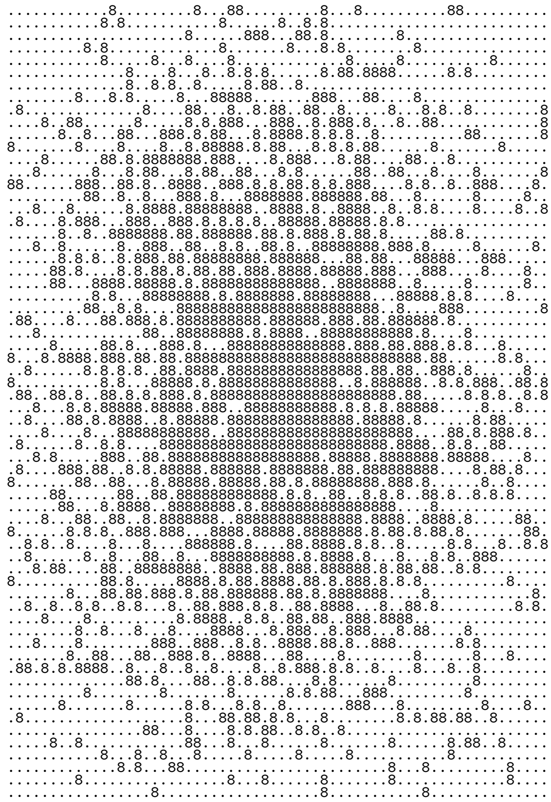
3.2 Fonctionalités à écrire

Ecrire les fonctions et procédures (elles ne sont pas données par ordre croissant de difficulté) :

- de construction d'images (construit_blanc() et construit_noir() rendent une image blanche, resp. noire à partir de rien, construit_composee(i1,i2,i3,i4) construit un image composée dont les 4 sous-images sont i1,i2,i3,i4)
- d'affichages en modes normal et profondeur.
- est-noire et est-blanche qui testent si l'image est noire, resp. blanche (.BBB.BBB.BBBB est blanche)
- copie
- aire (de la zone noire en considérant que le carre est de côté 1, aire(.NBN.NBNB)= 0.625)
- meme_dessin qui teste si deux images représentent le même dessin (.NBN.BBB.NNNN et ..NNN.NNN.NNNNBN.B.BBBBBN ont le même dessin)
- difference qui prend deux images et rend une image qui est noire exactement là où les 2 images sont de couleurs différentes
- rendmemoire qui rend tous les champs d'une image à la mémoire
- de lecture au clavier
- CompteSousImagesPleines(image,hauteur). Une image pleine de hauteur h est une image dont toutes les feuilles sont à profondeur h. Par exemple ..BBNB.NNBN.BBBN.NNNB est pleine de hauteur 2.
  Et . ..BBNB.NNBN.BBBN.NNNB N .NBN.BBNB .BNB ..BBNB.NBBN.BNBN.NBNB a deux sous images pleines de hauteur 2.
- arrondit qui prend une image et un entier k et arrondit aux pixels de taille $1/2^k$
  `arrondit`(image,0) transforme une image en blanc ou noir,
  `arrondit`(. N .BBNB B .N[.NNB(.NBNN)]BN,2) arrondit l'image en . N .BBNB B .NNBN
- négatif (procédure qui TRANSFORME une image en son négatif)
- simplifie qui simplifie la représentation d'une image (simplifie(. . N N .NNNN N .BBBB N .BNBN) TRANSFORME l'image en .NBN.BNBN)
- affichage 2^k pixel. Note : il est interdit d'utiliser un tableau de taille variable, ou tout autre utilitaire évolué (bibliothèque d'affichage graphique) pour l'affichage pixel.
- Alea qui prend en argument la profondeur k, et un entier n et qui rendra une image dont la partie noire sera constistuée de n pixels noirs à profondeur k, positionnés aléatoirement. Chaque image pouvant sortir de préférence avec équiprobabilité.
  Par exemple, Alea(1,2) rendra chacune des images
  .NNBB .BBNN .NBNB .BNBN .BNNB .NBBN,
  avec probabilité 1/6
  tandis que Alea(4,13) pourrait rendre

```
................
.8..............
.........8..8...
.....8..........
................
................
.....8......8...
................
............8...
.......8........
...8............
................
.........8......
...8...........8
................
.........8......
```

- nebuleuse qui prend k en argument et rend une image de profondeur k qui ressemble à une nébuleuse : la couleur de chaque pixel est tirée aléatoirement de telle manière que la densité de noir varie de quasiment 1 au centre à quasiment 0 aux angles. Par exemple, nebuleuse(6) pourrait donner



3.3 Quelques fonctionnalités plus compliquées (et non demandées)

Ceux qui souhaitent aller au-delà du projet peuvent tenter de faire :

- Zoom (algorithmique un peu technique)

  Les coordonnées d'un point sont données par son abscisse et son ordonnée, que l'on supposera être de la forme $n/2^k$ (On acceptera donc 5/8 mais pas 1/3). Le point inférieur gauche a donc pour coordonnées $(0, 0)$, le centre de l'image a pour coordonnées $(1/2, 1/2)$, etc.

  On représentera ces nombres la paire des entiers n et k (ce qui évitera les problèmes d'approximation sur les réels)

  La fonction Zoom prend en argument une image, une abscisse x et une ordonnée y, et rend la sous-image obtenue sur un carré de taille $1/2 * 1/2$ dont le point inférieur gauche est en (x, y). Par exemple Zoom(.NBBN , 3/8, 1/4) donne . .NBNB B .BNBN N .Ce qui déborde de l'image sera blanc, Par exemple Zoom(N , 1/4. 7/8) donne . B B .BBNN .BBNB

- ComposantesConnexes (algorithmiquement intéressant, les versions les plus efficaces peuvent être très subtiles)

  rend le nombre de CC de la partie noire CC(. ..NNBB .NNBN .NNBN .BNBN) rend 2 par exemple. CC(. B .BNNB N B) rendra 1 ou 3 suivant que l'on considère que la partie noire est un fermé ou un ouvert et que donc la connexité passe ou non par les coins.

- Compresse (algorithmiquement intéressant, types de données évolués à utiliser si vous voulez être efficace)

  transforme l'arbre en graphe acylique en utilisant un seul struct pour des sous arbres identiques. Exemple . .BNBN . B .BNBN N N .BNBN . BBB . B .BNBN N N utilisera 4 struct non noirs et 1 noir au lieu de 8 struc non noirs et 12 noirs

- Chute (algorithmiquement intéressant, un peu difficile, subtil)

  la gravité agit sur les pixels qui tombent et s'empilent dans le bas de l'image. Chute calcule le résultat.
  Exemple Chute (. .BNB.BBBN .BBNN B .NBBN) donne . B B .B.BBBNBN N