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 fonctions en C, qui vous feront manipuler explicitement récursivité, pointeurs, listes chaînées, pasages par adresse, etc. et poseront quelques 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 la formule $e = \sum_{n=0}^{\infty} 1/n!$.

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

- Implémentez la fonction $Puissance(x, n)$, x réél, n entier éventuellement négatif (On conviendra que $0^0 = 1$). Utilisez-la pour calculer 1.1^{10}, 1.01^{100}, 1.001^{1000}... $(1 + 10^{-k})^{10^k}$...

  Comparez les différentes méthodes : temps de calcul, mémoire, précision du résultat.
  Comparez la différence de précision selon que l'on utilise des "float" ou des "double";

  Sachez que $(1 + \epsilon)^{1/\epsilon}$ vaut environ

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

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

  Que valent les premières valeurs de A(m,0) ?

- 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

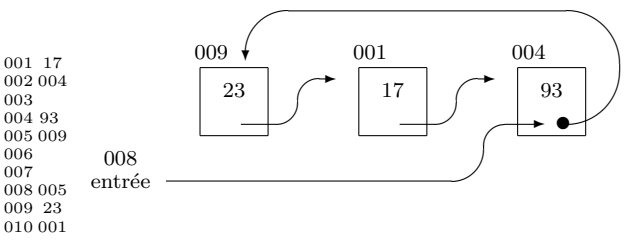
• • Un mini-programme avec en-tête, déclarations et quelques fonctionnalités est founi en annexe. Vous êtes invités à le compléter, en implémentant les fonctions et procédures suivantes (issues pour la plupart des partiels de l'an dernier) :

- **ZeroEnDeuxiemePosition** qui prend en argument une liste et rend vrai si et seulement elle a un 0 en deuxième position. Elle rendra donc vrai sur $[2, 0, 3, 6, 0]$ et faux sur $[0, 2, 0, 6, 0]$
- **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).
- **Compte0Initiaux** qui prend une liste et rend le nombre de 0 qui se trouvent en début de liste.
  Exemples: Compte0Initiaux([0,0,0,4,6,0,0,7]) rendra 3,
  Compte0Initiaux([1,0,0,0,4,6,0,0,7,0,0]) rendra 0, Compte0Initiaux([0,0,0,0,0]) rendra 5

  Donnez quatre versions de cette fonction :
  • Une récursive sans sous-fonctionnalité (et non terminale)
  • Une itérative
  • Une utilisant une sous-fonction récursive terminale avec argument supplémentaire in.
  • Une utilisant une sous-procédure récursive terminale avec argument supplémentaire inout.
- **IntersectionTriee** qui prend en argument deux listes $l1$ et $l2$, supposées triées dans l'ordre croissant, et rend la liste triée des éléments communs aux deux listes. Si x apparaît plusieurs fois, à savoir n_1 et n_2 fois, dans $l1$ et $l2$, alors x apparaîtra $min(n_1, n_2)$ fois dans $IntersectionTriee(l1, l2)$. Exemple, si $l1 = [2, 2, 2, 4, 4, 6, 8, 9]$ et $l2 = [1, 1, 2, 2, 3, 3, 4, 4, 8, 8]$ alors $IntersectionTriee(l1, l2)$ rendra $[2, 2, 4, 4, 8]$.
- **ElimineKpremiersX** qui prend une liste d'entiers l, un entier k et un entier x et élimine les k premières occurences de x dans la liste. S'il y a moins de k occurences de x dans l, alors elles sont toutes éliminées. Ne faire qu'un seul parcours de la liste.
- **ElimineKderniersX** qui prend une liste d'entiers l, un entier k et un entier x et élimine les k dernières occurences de x dans la liste. S'il y a moins de k occurences de x dans l, alors elles sont toutes éliminées. Ne faire qu'un seul parcours de la liste.
- La fonction **Permutations** en utilisant la technique "diviser pour régner" du cours.

  Ajoutez un compteur pour compter le nombre de malloc effectués. Commentez. Pouvez-vous réduire l'espace mémoire consommé ?

• • 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

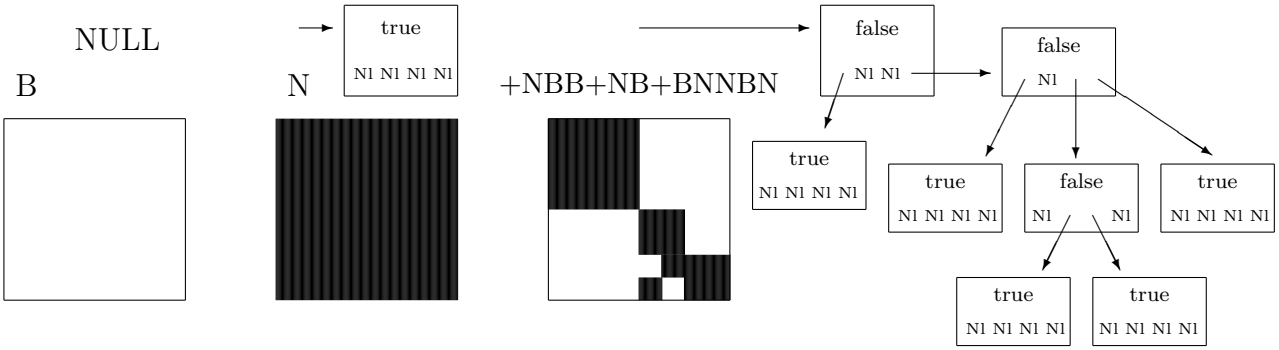
On représentera ces images avec la structure suivante :

```
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 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 préfixe pour les entrées sorties. 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

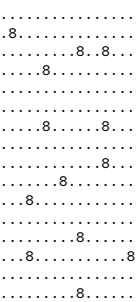
- En mode 2^k-pixel, l'affichage se fait sur 2^k lignes et 2^k colonnes, en utilisant le point pour le blanc, le 8 pour le noir, et le $-$ quand la résolution de l'affichage est insuffisante pour donner une couleur.
  L'affichage 2^3-pixel de + N +BBNB B +N+NNB+NBNN BN donne :

```
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) :

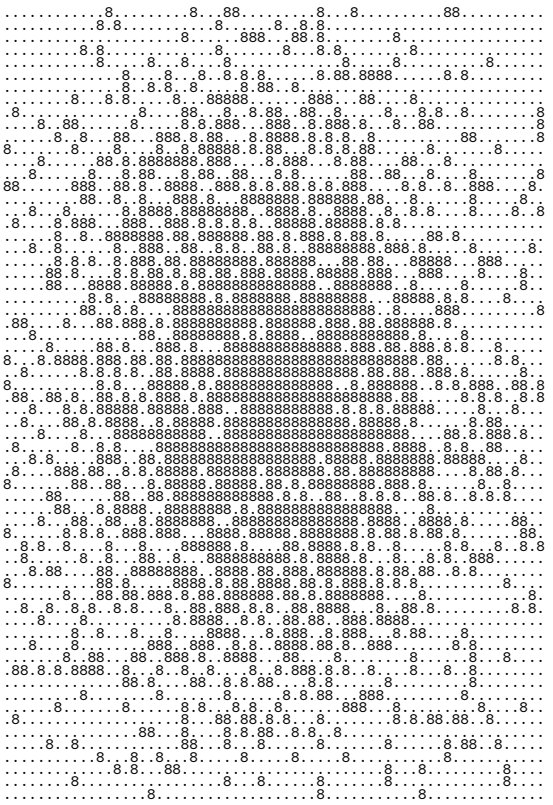
- de construction d'images : Construit_blanc() et Construit_noir() rendent une image blanche, resp. noire à partir de rien. Construit_composee(ihg, ihd, ibg, ibd) construit une image composée dont les 4 sous-images sont ihg, ihd, ibg, ibd.
- d'affichages en modes normal et profondeur.
- EstNoire et EstBlanche qui testent si l'mage en argument est noire, resp. blanche (+BBB+BBB+BBBB est blanche)
- Copie qui rend une nouvelle image avec de nouveau blocs mémoire ayant la même structure que l'image en argument
- Aire qui rend le taux de noir de l'image argument, Aire(+NBN+NBNB)= 0.625
- Rendmemoire qui rend tous les blocs d'une image à la mémoire
- Lecture qui rend une image à partir des caractères tapés au clavier
- CompteSousImagesGrises(image). Une image grise est une image dont l'aire est comprise entre 1/3 et 2/3. La fonction rendra le nombre de sous images grises. L'image + N +BNBB B +NBBB est grise. L'image + B N + N +BNBB B +NBBB + N +BNBB B +NNBB 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)
- Negatif, procédure qui TRANSFORME l'image argumen en son négatif
- UnionNoire qui prend deux images en argument et rend vrai ssi leur superposition est noire. Si les images sont + +NBNB +BNNB B +NNBB et + N +NBNN N +NBNB, la fonction rendra faux car la superposition n'est pas noire sur le pixel en bas à droite.
- Intersection qui prend deux images en argument. La deuxième n'est pas modifiée. La première est modifiée, elle ne restera noire que là où les deux images sont noires. intersection de + +NBNB +BNNB B +NNBB et de + N +NBNN N +NBNB tranformera la première en + +NBNB +BBNB B +NBBB
- Affichage2Pixel. Note : Il est largement préférable de ne pas utiliser un tableau de taille variable (4^k par exemple). il est inerdit d'utiliser tout autre utilitaire évolué (bibliothèque d'affichage graphique) pour l'affichage pixel.
- Alea qui prend en argument une 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



3.3 Quelques fonctionnalités plus compliquées (et non demandées)

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

- 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



- 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