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 25 novembre à 8h
- prérendu de la partie 3 le 9 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 ?

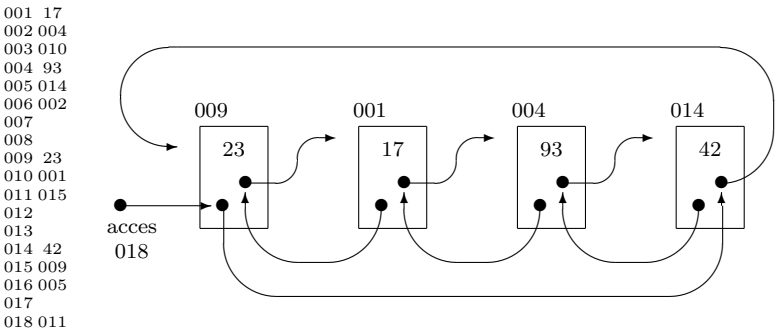
- Les ListesZ sont construites avec des blocs. Chaque bloc possède trois champs, un champs `valeur` de type entier, un champs `next` qui pointe vers le bloc suivant, et un champs `prec` qui pointe vers le champs `next` du bloc précédent. S'il n'y a qu'un seul bloc, les blocs suivant et précédent sont 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.

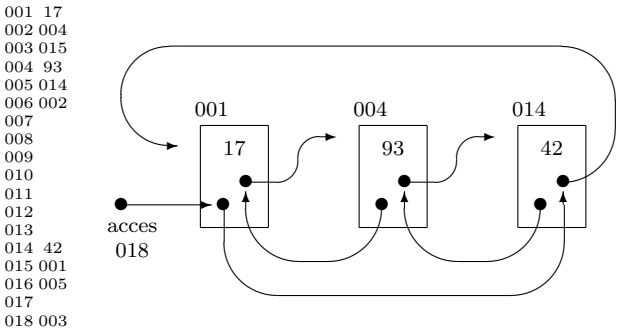
  Écrire le code de la procédure `ZElimine` qui prend en argument un pointeur `acces` et enlève le bloc à l'intérieur duquel `acces` pointe, qui sera rendu à la mémoire.

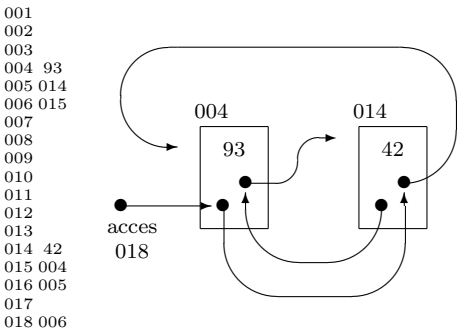
  Après l'appel, `acces` à l'intérieur du bloc qui suivait celui qui a été éliminé, sous réserve que celui-ci existe.

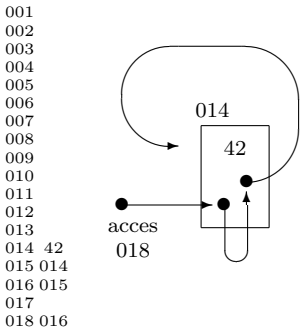
  Si la liste est vide, `Zelimine` ne fait rien (elle ne plante pas)

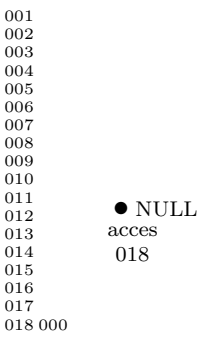
  Exemple, si la liste contient 23, 17, 93, 42 selon le premier schéma ci-dessous, et que l'on effectue plusieurs fois `Zelimine`, on obtient successivement les listes ci-dessous.











3 Arbres : Quadtrees

L'énoncé sera fourni plus tard.

4 Appendice : la suite Yn

L'explication de ce qui se passe pour l'exercice 1.2 sera fournie plus tard.