Programmation C avancée
TP 1

Les exercices sont indépendants et de difficulté variable. Prenez l'habitude de classer vos TP et aussi d'organiser les sources de manière rigoureuse. Par exemple, les fichiers de code peuvent raisonnablement être rassemblés dans un fichier `sources` et vos fichiers d'entêtes (lorsqu'il est intéressant de les créer) dans un fichier `include`.

Prenez aussi l'habitude de rédiger des fichiers `Makefile` pour vos TP contenant les régles pour compiler tous vos exercices. Sachant que l'utilitaire `make` peut étre récursif, une organisation possible est de procéder comme il suit:

```
# Makefile TP1
all:
      make mytee
      make exo2
      make exo3
      make ...

mytee: ....
      gcc -o mytee ...

exo2: ....
      gcc ...
```

Pour tous vos programmes utilisant de l'allocation dynamique (`malloc`), veuillez à bien utiliser l'utilitaire `valgrind` pour vérifier que vous libérez bien la mémoire allouée par vos programmes.

Pour tous les exercices (et pour tous les TP de manière générale), il vous sera demandé de produire un code propre, très propre. Ainsi, la gestion des erreurs est très importante et leur report est très important. Vous devez ainsi utiliser les valeurs de retour de `malloc`, `fopen`, `printf`, ... et afficher sur la sortie standard les messages d'erreur correspondants.

Toutes les règles énoncées plus haut ne seront pas rappelées à chaque énoncé mais les bonnes habitudes de code seront considérées comme acquises.

Exercice 1 Un peu d'entrées/sorties et intégration dans `Unix`

Une des fonctionnalités célébres d'`Unix` est la fonction `tee`. C'est la fonctionnalité utilisée pour générer des fichier de `log`. L'esprit d'`Unix` est de proposer des briques de bases simples pouvant s'enchaéner dans des pipes (caractére |) et ainsi résoudre un problème plus complexe.

Par exemple, `cat fichier1 fichier2` concaténe les textes des deux fichiers et affiche le résultat sur la sortie standard. `wc -l fichier` compte le nombre de lignes dans le fichier et affiche le résultat dans la sortie standard.
Maintenant, la commande `cat fichier1 fichier2 | wc -l` concatène les deux fichiers mais au lieu d'afficher le résultat, ce dernier est passé en argument de la commande suivante après

le pipe. Ainsi cette commande complexe compte le nombre de lignes dans la concaténation du texte des deux fichiers `fichier1` et `fichier2`.

- écrire un programme dont le nom d'exécutable sera `mytee` qui recopie ce qu'il lit dans l'entrée standard à la fois sur la sortie standard mais aussi dans un fichier dont le nom est donné en argument de l'exécutable.

De manière générale, `tee` fabrique un fichier de log comme il suit :
`commande1 argument1 | tee resultat_partiel.log | commande2`
(il peut y avoir bien plus de commande chaînée avant ou après). Le `tee` sauvegarde le résultat partiel obtenu après la première commande dans un fichier mais laisse aussi passer le résultat pour des traitements futurs.

- Utilisez votre exécutable `mytee` pour compter le nombre de lignes de la page du manuel `Unix` de la fonction `fprintf` tout en sauvegardant le contenu du manuel dans un fichier `doc.log`.

Exercice 2 Autour des coefficients binomiaux

Les nombres binomiaux peuvent être construits avec le triangle de Pascal.

		p							
		0	1	2	3	4	5	6	7
	0	1							
	1	1	1						
	2	1	2	1					
n	3	1	3	3	1				
	4	1	4	6	4	1			
	5	1	5	10	10	5	1		
	6	1	6	15	20	15	6	1	
	7	1	7	21	35	35	21	7	1
		...							

Par récurrence, on peut aussi définir les nombres binomiaux $b(n, p)$ par les conditions :

- Pour tout n entier positif : $b(n, 0) = b(n, n) = 1$.
- Pour tout $n > 0$ et pour tout $0 < p < n$: $b(n, p) = b(n-1, p) + b(n-1, p-1)$

On essaie maintenant d'expérimenter le calcul des binomiaux avec le langage C.

- Programmer une fonction C fortement récursive pour calculer $b(n, p)$.
- Estimer (soit par le calcul, soit par l'expérimentation) la complexité du calcul d'un coefficient binomial avec la méthode récursive.
- Reprogrammer le calcul des binomiaux avec un tableau en utilisant la relation de récurrence rappelée plus haut. Attention à la gestion de la mémoire.

- Si vous avez grand appétit, faites des bancs d'essais en collectant les temps de calcul des deux méthodes.

Exercice 3 Exécutable à arguments variables

Un problème issu des petites classes consiste à calculer la somme de plusieurs fractions. Pour cela, les fractions doivent être réduites au même dénominateur. Dans le but de minimiser les calculs, il est astucieux de choisir un nombre le plus petit possible tel que tous les dénominateurs divisent ce nombre (c'est le plus petit multiple commun).

- Produire un exécutable qui retourne le plus petit multiple commun des entiers donnés en argument dans la ligne de commande.

Par exemple, vous devriez avoir:

```
./ppcm 12 12 12 12
12
./ppcm 1
1
./ppcm 12 6 15
60
./ppcm 12 34 54 72 28 72 81 18 26
1002456
```

Exercice 4 Pour gagner du temps sur l'avenir

Profiler les autres exercices avec l'utilitaire `Unix` `gprof`. Ce dernier est capable notamment de compter le nombre d'appels à chaque fonction appelée par vos programmes. Pour cela, il faut compiler vos sources avec l'option `-pg` (il suffit donc d'éditer votre Makefile). Il faut ensuite exécuter le programme à profiler avec des arguments convenables puis enfin demander le résumé en console via

```
gprof nom_executable gmon.out
```