

Projet de « Programmation en langage C »

Licence bio-informatique

Année 2001-2002

1 Modalités de réalisation du projet

Trois sujets de projet sont proposés au choix, à réaliser en langage C. Les sujets 1 et 2 sont à réaliser par groupes de 3 étudiants et le sujet 3 par groupes de 2 étudiants (au maximum).

- sujet 1 : réalisation d’une bibliothèque de fonctions permettant l’évaluation d’expressions arithmétiques utilisant des entiers arbitrairement grands (section 2) ;
- sujet 2 : réalisation d’un compresseur de fichiers par l’algorithme de Huffman (section 3) ;
- sujet 3 : réalisation d’une commande de recherche de motifs dans les lignes de fichiers (section 4).

Le projet fera l’objet d’une soutenance orale le **21 janvier 2002** dans laquelle chaque membre d’un groupe devra exposer le travail qu’il a effectué. Un rapport sera remis au plus tard le **11 janvier 2002**. Il doit exposer brièvement (une dizaine de pages maximum) les structures de données utilisées, les algorithmes choisis et les difficultés rencontrées. Il ne s’agit pas de commenter le code une seconde fois, ni d’expliquer chaque fonction en détail (c’est le rôle, à nouveau, des commentaires du source, en particulier de ceux se trouvant devant les prototypes des fonctions), mais d’en présenter une synthèse. Enfin, le source devra également être fourni par mail au plus tard pour le 18 janvier.

Les points suivants devront faire l’objet d’une attention particulière lors de l’écriture du programme : sa correction, sa lisibilité et sa clarté, sa modularité (développement de petites fonctions, répartition du source en plusieurs fichiers regroupant des fonctions de façon logique), sa portabilité, ainsi que le choix des structures de données et l’efficacité des algorithmes. Ce dernier point n’est cependant pas essentiel ici, car il fait l’objet de cours ultérieurs (algorithmique, et automates pour le sujet 3). L’utilisation de `make` et `gdb` sera utile pour accélérer le développement.

2 Sujet 1 : calculs sur entiers en précision arbitraire

2.1 Objet du projet

Le projet consiste à écrire une bibliothèque de fonctions permettant la manipulation d’entiers arbitrairement grands, ainsi qu’un programme permettant à un utilisateur de tester cette bibliothèque de façon interactive et pratique. On pourra supposer, pour simplifier les calculs et adopter les types de données indiqués plus bas, que le plus grand entier de type `unsigned long int` sur la machine où tournera le programme est supérieur à 9999×9999 . Cette hypothèse, en général correcte, devra néanmoins être vérifiée par les fonctions de la bibliothèque.

2.2 Représentation des entiers non signés (positifs ou nuls)

Un entier non signé en précision arbitraire sera codé en base 10000. On remarquera que coder un nombre en base 10000 consiste à l’écrire d’abord en base 10, puis à regrouper les chiffres quatre par quatre en commençant par la droite. Les « chiffres » de la base 10000 sont ainsi les entiers de 0 à 9999. Un entier n sera représenté par une liste chaînée. Chaque élément de cette liste contiendra un « chiffre » du codage de n en base 10000, *i.e.*, un nombre compris entre 0 et 9999.

Afin de simplifier le développement des opérations arithmétiques, le codage commencera par la droite. Ainsi, le premier élément de la liste code le chiffre des « unités » en base 10000, c’est-à-dire la valeur de n modulo 10000 (ou, dit différemment, regroupe les chiffres des unités, des dizaines, des centaines et des milliers dans l’écriture en base 10 de n).

La liste vide représentera l’entier zéro. Les listes seront normalisées : la dernière cellule d’une liste devra contenir un chiffre non nul. La figure suivante donne quelques exemples de codages d’entiers.

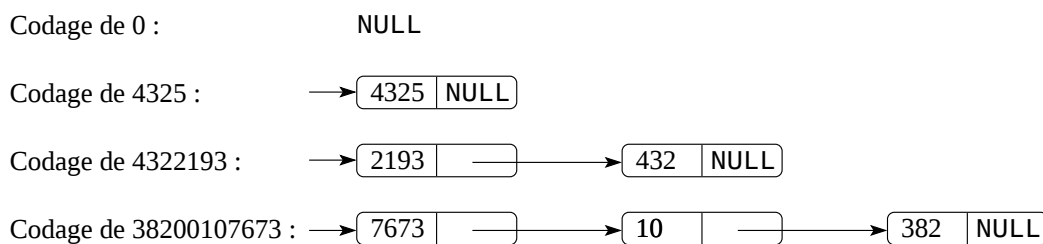


FIG. 1 – Exemples de représentation d'entiers par des listes chaînées

2.3 Représentation des entiers signés

Un entier signé en précision arbitraire sera représenté par son signe et sa valeur absolue, représentée comme ci-dessus par une liste chaînée.

2.4 Travail demandé

A. On demande de réaliser une bibliothèque contenant :

- A.1. un type `grand_entier` permettant de coder les entiers signés arbitrairement longs ;
- A.2. une fonction `grand_entier nouvel_entier(char *)` ; construisant un `grand_entier` à partir de sa représentation sous forme de chaîne de caractères ;
- A.3. une fonction `int afficher_grand_entier(grand_entier)` ; permettant d'afficher un tel entier ;
- A.4. une fonction `int est_nul(grand_entier)` ; de comparaison à zéro ;
- A.5. une fonction `int est_positif(grand_entier)` ; testant si un entier est positif ;
- A.6. les fonctions usuelles de comparaison (`egal_a`, `different_de`, `inferieur_a`, `inferieur_ou_egal_a`, `superieur_a`, `superieur_ou_egal_a`) prenant en argument deux `grand_entier`, les comparant, et renvoyant 1 ou 0 selon que la comparaison est vraie ou fausse ;
- A.7. des fonctions

- `grand_entier somme(grand_entier, grand_entier)` ;
- `grand_entier difference(grand_entier, grand_entier)` ;
- `grand_entier produit(grand_entier, grand_entier)` ;
- `grand_entier quotient(grand_entier, grand_entier)` ;
- `grand_entier reste(grand_entier, grand_entier)` ;
- `grand_entier pgcd(grand_entier, grand_entier)` ;
- `grand_entier ppcm(grand_entier, grand_entier)` ;

calculant la somme, la différence, le produit, le quotient et le reste de la division euclidienne, le plus grand diviseur commun et le plus petit multiple commun, respectivement, de leurs deux arguments ;

- A.8. une fonction `grand_entier factorielle(unsigned long)` ; calculant $n! = n \times (n-1) \times \dots \times 3 \times 2$. L'argument de cette fonction sera donné sous forme d'un `unsigned long int` ;

- A.9. une fonction `grand_entier cnp(unsigned long, unsigned long)` ; calculant $C_n^p = \frac{n!}{p!(n-p)!}$.

Les arguments de cette fonction seront donnés sous forme de deux `unsigned long` ;

- B. Écrire une interface utilisateur pour permettre un test (convivial) des fonctions ci-dessus.
- C. Écrire un en-tête `grands_entiers.h` donnant accès au prototypes, constantes et variables fournies par la bibliothèque ;
- D. Compiler la bibliothèque dynamique sous le nom `libbignums.so`. Avec `gcc` sur `nivose`, on pourra utiliser une ligne de commande du type (info `gcc` pour plus de détails!) :

```
$ gcc -Wall -pedantic <fichiers source> -o libbignums.so -shared -fpic
```

Compiler enfin l'interface qui fait appel à cette bibliothèque de la façon suivante :

```
$ gcc -Wall -pedantic -L<repertoire bibliothèque> interface.c -o interface -lbignums
```

À l'exécution l'interface, la variable `LD_LIBRARY_PATH` spécifiera le répertoire contenant la bibliothèque.

3 Sujet 2 : compression en codage de Huffman

3.1 Objet du projet

Le but de ce projet est d'écrire un programme permettant de compresser des fichiers, sans perte d'information, afin qu'ils occupent moins d'espace sur disque. On demande également d'écrire un décompresseur qui devra restaurer le fichier original. Les algorithmes de compression sont nombreux. Celui proposé ici est l'algorithme de Huffman.

3.2 Principe de l'algorithme de Huffman

Une idée apparue très tôt en informatique pour compresser les données a été exploitée indépendamment dans les années 1950 par Shannon et Fano. Elle est basée sur la remarque suivante : les caractères d'un fichier sont habituellement codés sur un octet, donc tous sur le même nombre de bits. Il serait plus économique en terme d'espace disque, pour un fichier donné, de coder ses caractères sur un nombre variable de bits, en utilisant peu de bits pour les caractères fréquents et plus de bits pour les caractères rares. Le codage choisi dépend donc du fichier à compresser. Les propriétés d'un tel codage sont les suivantes :

- a) chaque caractère est codé sur un nombre différent de bits (pas nécessairement un multiple de 8) ;
- b) les codes des caractères fréquents dans le fichier sont courts ; ceux des caractères rares sont longs ;
- c) bien que les codes soient de longueur variable, on peut décoder le fichier compressé de façon unique.

La dernière de ces trois propriétés est automatiquement assurée si l'on a la propriété suivante :

- d) si c_1 et c_2 codent deux caractères différents, c_1 ne commence pas par c_2 et c_2 ne commence pas par c_1 .
- En effet, si la propriété d) est assurée, lorsqu'on décode le fichier compressé en le lisant linéairement, dès que l'on reconnaît le code d'un caractère, on sait que l'on ne pourra pas le compléter en un autre code.

L'algorithme de Huffman, qui garantit ces propriétés, fonctionne de la façon suivante :

- on calcule tout d'abord les fréquences d'apparition de chaque caractère dans le fichier à compresser ;
- on calcule ensuite pour chaque caractère un code satisfaisant les propriétés a), b), d).
- on écrit ce code en début de fichier compressé (pour que le décompresseur y ait accès), suivi des données compressées elles-mêmes.

Pour calculer le code de chaque caractère, l'algorithme construit un arbre binaire de haut en bas :

- i) les feuilles de l'arbre sont les lettres apparaissant dans le fichier ;
- ii) deux nœuds n_1 et n_2 de fréquences minimales sont choisis. On construit un nouveau nœud n qui devient père de n_1 et n_2 , et dont la fréquence est la somme de celle de n_1 et celle de n_2 ;
- iii) on répète l'étape précédente jusqu'à atteindre une unique racine.

Exemple On suppose que le fichier à compresser contient le texte **Une banane** (sans fin de ligne). Les fréquences des caractères sont données figure 2. Comme caractères ayant une fréquence minimale, on peut choisir ' ' et 'U' (il y a d'autres choix possibles). On obtient l'arbre de la figure 3. Le nœud ajouté a comme fréquence $1 + 1 = 2$. À ce stade, le nœud de fréquence minimale est 'b' (fréquence 1), mais il y a trois choix pour l'autre nœud de fréquence minimale (fréquence 2) avec lequel regrouper 'b' : on peut choisir l'une des feuilles 'a' ou 'e', ainsi que le nœud construit à l'étape précédente. Ce dernier choix est illustré figure 4. Le nœud ainsi créé a pour fréquence $2 + 1 = 3$. Une séquence de choix est indiquée ci-dessous.

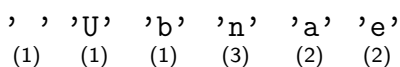


FIG. 2 – Fréquences des caractères

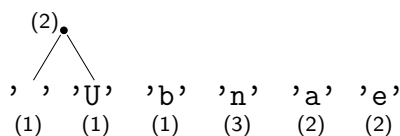


FIG. 3 – 1^{ère} étape

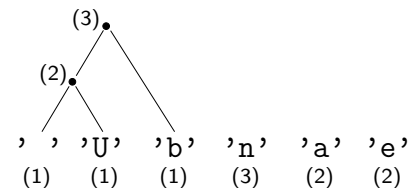


FIG. 4 – 2^{ème} étape

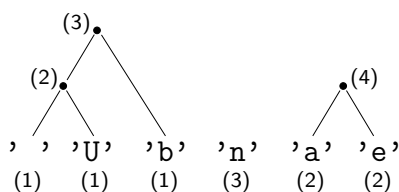


FIG. 5 – 3^{ème} étape

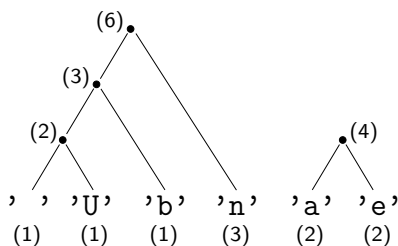


FIG. 6 – 4^{ème} étape

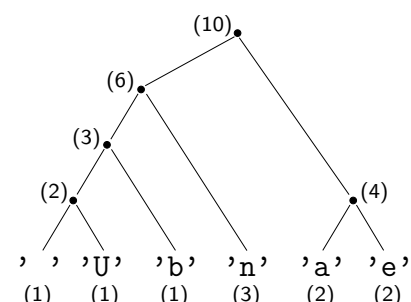


FIG. 7 – Un arbre final possible

Avec d'autres choix, on peut obtenir des arbres différents, mais l'algorithme fonctionne avec tout choix respectant la construction ci-dessus. À partir de l'arbre, on construit le code d'un caractère c en lisant le chemin qui va de la racine à c , un pas à gauche étant lu comme 0 et un pas à droite comme 1. Dans l'exemple précédent, on obtient la table suivante et la suite de bits 0001011100000011001100111 comme code de *Une banane*.

Caractère	' '	'U'	'a'	'b'	'e'	'n'
Fréquence	1	1	2	1	2	3
Code donné par la figure 7	0000	0001	10	001	11	01

On peut voir que ce codage vérifie toujours les propriétés *a)* à *d)*, et permet donc la compression et la décompression. Avec cet algorithme, le décompresseur doit connaître les codes construits par le compresseur. Lors de la compression, le compresseur écrira donc ces codes en début de fichier compressé, sous un format à définir, connu du compresseur et du décompresseur. Le fichier compressé aura donc deux parties disjointes :

- une première partie permettant au décompresseur de retrouver le code de chaque caractère ;
- une seconde partie contenant la suite des codes des caractères du fichier à compresser.

Attention, le décompresseur doit toujours pouvoir trouver la séparation entre ces deux parties.

Un dernier problème technique est que le nombre de bits occupés par le codage du fichier n'est plus nécessairement un multiple de 8. Une solution est de calculer l'arbre et tous les codes comme s'il existait un 257^{ème} caractère (appelé ici pseudo-caractère) marquant la fin du fichier original. Ce pseudo-caractère aura donc comme fréquence 1. Lorsque la fin du fichier original est atteinte, le compresseur écrit le code de ce pseudo-caractère, et complète par des bits 0 pour avoir une taille totale du fichier multiple de 8 bits. Lorsque le décompresseur rencontre le code de ce caractère, il sait que la fin de fichier compressé est atteinte.

3.3 Travail demandé

- La première partie du travail consiste à écrire une bibliothèque permettant des entrées-sorties bit à bit.
 - Définir un type `bit` permettant de représenter un bit.
 - Définir un type `fic_bin` ainsi que des fonctions


```

          - fic_bin* open_fic_bin(char *nom, char *mode);,
          - int close_fic_bin(fic_bin *f);,
          - int write_fic_bin(fic_bin *f, bit b);,
          - int read_fic_bin(fic_bin *f, bit *b);
          
```

 permettant d'effectuer des entrées-sorties dans un fichier bit à bit. La structure `fic_bin` représentera un fichier ouvert et permettra en plus de mémoriser les bits dont on aura demandé la lecture ou l'écriture, mais en nombre inférieur à 8, donc ne formant pas encore un octet complet.
 - Compiler cette bibliothèque dynamique sous le nom `libficbin.so` (cf. section 2.4 partie D pour la compilation et l'utilisation de la bibliothèque).
- Écrire une fonction permettant de calculer les fréquences des caractères apparaissant dans un fichier, puis implémenter l'algorithme de calcul de l'arbre.
- Écrire les fonctions de compression et de décompression utilisant les fonctions de `libficbin.so`.
- Écrire une commande utilisateur permettant de compresser et de décompresser les fichiers.

4 Sujet 3 : recherche de motifs

4.1 Objet du projet et travail demandé

Le projet consiste à écrire un programme dont l'exécutable, appelé ici **mygrep** a pour synopsis :

```
mygrep [ -[[AB] ]num ] [ -[bchiLlnsv] ] [ -e ] motif [ [ -e ] motif... ] [ fichiers... ]
```

Il permettra d'effectuer des recherches de motifs dans les lignes de fichiers. Un motif est une chaîne de caractères. Chaque ligne de fichier contenant l'un des motifs s'appelle une *correspondance*. Sans option, le programme recherche toutes les correspondances, puis :

- si le fichier contient au moins un caractère nul (`'\0'`), il est considéré comme un fichier non imprimable, et le programme se contente d'indiquer si oui ou non une correspondance a été trouvée dans le fichier (mais sans afficher la ou les lignes contenant le motif) ;
- si le fichier ne contient aucun caractère nul, chaque correspondance est affichée sur la sortie standard.

Enfin, si aucun nom de fichier n'est donné, **mygrep** lira ses données sur l'entrée standard.

4.2 Motifs particuliers

Les motifs acceptés par **mygrep** sont des chaînes de caractères. L'accent circonflexe `^` et le symbole dollar `$` ont un rôle spécial : ils correspondent respectivement à une chaîne vide au début et en fin de ligne. Par exemple les correspondances du motif `^programme` sont les lignes commençant par le mot **programme**. Les correspondances du motif `^programmation$` sont les lignes ne contenant que le mot **programmation**.

Par ailleurs, on appelle *mot* toute séquence composée de lettres, de chiffres et du caractère souligné `_`. Les séquences `\<` et `\>` indiquent respectivement à une chaîne vide en début et en fin de mot. Ainsi, si le motif est `\<mot\>`, la ligne

Un mot de 3 lettres.

est une correspondance, mais pas la ligne

Un motif plus long...

4.3 Options du programme

Les options suivantes doivent permettre de changer le comportement du programme. On pourra se servir de la fonction de bibliothèque `getopt()` pour analyser les options tapées par l'utilisateur. Même si l'une des 3 premières options est présente, **mygrep** n'affichera chaque ligne du fichier qu'au plus une fois.

Option	Effet
<code>-num</code>	afficher autour de chaque correspondance les num lignes se trouvant avant et après dans le fichier.
<code>-A num</code>	afficher avant chaque correspondance les num lignes se trouvant avant la correspondance dans le fichier.
<code>-B num</code>	afficher après chaque correspondance les num lignes se trouvant après la correspondance dans le fichier.
<code>-b</code>	préfixer chaque ligne de sortie par sa position en octets par rapport au début du fichier d'entrée.
<code>-c</code>	ne pas afficher les lignes sélectionnées, mais afficher le nombre de lignes de correspondance pour chaque fichier d'entrée. Avec l'option <code>-v</code> , afficher le nombre de lignes ne correspondant <i>pas</i> au motif.
<code>-e motif</code>	utiliser le motif indiqué. Cette option permet d'utiliser des motifs commençant par <code>-</code> , et d'indiquer plusieurs motifs.
<code>-h</code>	ne pas afficher le nom des fichiers dans les résultats lorsque plusieurs fichiers sont parcourus.
<code>-i</code>	ignorer les différences majuscules/minuscules dans le motif comme dans les fichiers d'entrée.
<code>-L</code>	ne pas afficher les correspondances, mais uniquement le nom des fichiers ne contenant aucune correspondance.

Option	Effet
-l	ne pas afficher les correspondances, mais uniquement le nom des fichiers contenant au moins une correspondance.
-n	préfixer chaque ligne de sortie par son numéro dans le fichier d'entrée.
-s	ne pas afficher les messages d'erreurs concernant les fichiers inexistantes ou illisibles.
-v	inverser la mise en correspondance : sélectionner les lignes ne correspondant pas au motif.