Rapport Projet 3 PIM:

Résumé du rapport (10 lignes max) :

-- A remplir pour le 15 janvier.

==========

Introduction du problème et contenu du rapport (sommaire) :

Le problème est le suivant : Donné un flot d'information (texte ou image par exemple), est-il possible de **compresser** ces informations **sans en perdre** grâce à un algorithme?

Le problème est donc celui de la **lossless compression** (ou compression sans perte) qui à émergé durant les années **1950**.

Nous procéderons à la compression via la **méthode d'Huffman** (allocation de plus de bits pour les caractères avec une faible occurrence) en l'implémentant sur un **fichier d'entrée binaire** afin de garantir l'universalité de la compression.

Sommaire :

1. Architecture en modules
2. Présentation des choix réalisés
3. Présentations des algorithmes et types de données
4. Démarche adoptée pour le test du programme
5. Difficultés rencontrées et solutions apportées
6. Organisation de l'équipe
7. Bilan technique et perspectives d'amélioration
8. Bilans personnels

==========

1. Architecture en modules :

Le programme de compression d'Huffman créé est divisé en plusieurs parties :

- Tout d'abord, nous disposons de **Compression.ads**, qui est le fichier de **spécification** du programme sous la **forme de raffinages**,

- Ensuite **Compression.adb** qui est l'**implémentation** des procédures et autres fonctions utilisées par l'algorithme de compression,

- Puis pour finir **Compresser.adb** qui est la pièce maîtresse du programme. Il s'agit du **programme à compiler** puis à appeler afin de compresser un fichier binaire.

==========

2. Présentation des choix réalisés :

L'algorithme de compression nécessitant un nombre et une diversité importante de données (caractères, code ASCII, code binaire, nombre d'occurrences), nous avons choisi de stocker ces dernières dans **2 types différents de tables de hachage.**
La première, issue du package TH prends :

- en clef, la **représentation binaire** du code du caractère suivant la table ASCII,
- en valeur, le **nombre d'occurrence** dans le document.

La seconde contient cette fois-ci les **représentations binaires** ET en **code de Huffman** afin de faciliter leur accès.

La table de hachage nous à paru être le meilleur choix car en prenant comme fonction de hachage l'emplacement dans la table ASCII de chaque caractère, nous pouvons accéder en un **temps constant** à toutes les données utiles pour la compression.

Pour ce qu'il en est du stockage de l'arbre de Huffman, nous avons choisi de créer une **pile** contenant un pointeur vers le dernier élément et un tableau contenant tous les nœuds et feuilles de l'arbre, toujours dans un souci d'**efficacité** et de **rapidité**.
Chaque noeud est constitué :

- d'un **symbole** (qui est une chaîne de caractère) valant "--------" pour les noeuds et la représentation binaire du code ASCII pour les caractères,
- du **nombre d'occurrences**,
- du **sous arbre droit** et **gauche**
- d'un "**historique**", étant un pointeur vers le noeud parent
- et d'un **booléen isSeen**, permettant de garder une trace de notre passage ou non par ce nœud lors de l'exploration de l'arbre.

==========

3. Présentation des algorithmes / types de données :

Présentation de l'algorithme permettant de récupérer les symboles et leur occurrence (GetSymbols) :

Nous parcourons chaque octet du **fichier binaire** à compresser avec une **boucle for** et nous stockons les symboles dans un table de hachage à l'emplacement correspondant au code ASCII du symbole.

- Si le caractère n'est pas apparu par avant, il est initialisé une entrée pour ce caractère,
- Sinon, nous mettons à jour son nombre d'occurrences.

Une fois l'entièreté du fichier lue, nous ajoutons à la table de hachage l'élément correspondant à l'**élément final**, le '\$' de code -1.

Présentation de l'algorithme de construction de l'arbre de Huffman (BuildHuffmanTree) :

Après avoir obtenu la table de hachage contenant les symboles et leur occurrences, nous créons un tableau de la taille du nombre de symboles différents. Ce tableau comportera des **pointeurs** correspondant aux **nœuds** (initialement aux **feuilles**) de l'arbre.

Ensuite, nous trions ce tableau à l'aide de la procédure **SortArray** et nous créons un nœud avec les deux sous arbres ayant les occurrences les plus faibles (la nouvelle occurrence est la somme des deux plus petites).

Nous réitérons alors ce procédé jusqu'à obtenir un seul arbre : notre **arbre de Huffman**.

Présentation de l'algorithme permettant d'obtenir le codage de Huffman des différents symboles (ExploreTree) :

Cette procédure est une **procédure récursive** qui a pour **critère d'arrêt** le fait que le nœud sur lequel nous somme ne possède pas d'enfant (c'est donc une feuille).

- Si tel est le cas, on enregistre dans la table de hachage des codes de Huffman le symbole et le code correspondant,
- Sinon, on appelle la procédure sur le fils gauche et on ajoute 0 puis sur le fils droit et on ajoute 1.

Présentation de l'algorithme de parcours infixe de l'arbre (InfixBrowsing) :

Cette procédure est aussi une **procédure récursive**.
Comme toute procédure récursive, il nous faut alors une **condition d'arrêt** et une structure récursive ou le rappel de la fonction permet de se rapprocher de sa **terminaison**.

Ici, la condition d'arrêt est **passée en argument dans l'appel** (il s'agit de "it"). It est un compteur que l'on incrémente à chaque fois que **InfixBrowsing** passe par une feuille. Ce compteur permet alors de savoir quand s'arrêter (en l'occurence, lorsque l'on à récupérer tous les symboles utilisés),

Et les différents appels dépendent de l'endroit où l'on se trouve dans l'arbre.

- Lorsque l'on se trouve sur une feuille, la fonction doit être rappelée afin de regarder le **prochain sous arbre disponible** (dans notre cas, le frère droit du nœud actuel).
- Lorsque l'on se trouve à un nœud ou les enfants ont déjà été parcourus, on rappelle **InfixBrowsing** sur le nœud **parent**.
- Lorsque le parcours infixe est continuable, alors on rappelle la fonction simplement sur le **fils gauche** du nœud actuel.

Ce parcours infixe de l'arbre nous permet entre autre de garder une certaine **cohérence** dans l'historique des symboles utilisés, nous permettant ensuite lors de la décompréssion de savoir quel code appartient à quel symbole **beaucoup plus facilement**.

==========

4. Démarche adoptée pour le test :

Tout d'abord, nous avons procédé à plusieurs tests faits à la main en comparant, après appel des différents algorithmes décrits plus haut, les résultats obtenus à ceux attendus.

Les fonctions principales étaient alors testées :

- **GetSymbols** devait renvoyer une table de hachage avec les caractères aux bons endroits,
- **BuildHuffmanTree** devait renvoyer le bon parcours infixe de l'arbre ainsi créé,
- et **GetTextCode**, devait quant à elle renvoyer une table de hachage permettant de **reconstruire** par la suite l'arbre de Huffman lors de la décompression.

Finalement, les tests étaient conclus par l'utilisation de la **commande diff** entre le fichier compressé obtenu et le fichier compressé attendu. Toutes ces vérifications nous ont ainsi amené à la conclusion que le **programme était robuste** et permettait effectivement de compresser un fichier binaire.

==========

5. Difficultés rencontrées ⇒ solutions apportées et justifications :

La majeure partie des difficultés s'est faite ressentir lors du **débugage** du fichier. Entre les différentes erreurs d'**appel**, d'**index**, de **pointeur** etc, les sources d'erreurs étaient multiples.

En effet, une fois les codes écrits, le plus dur était de **revenir** sur nos implémentations et porter un regard critique afin de comprendre d'où peut venir le problème et comment le régler.

Cette étape étant compliquée en étant rivé sur le code et en essayant simplement de trouver des erreurs, nous avons adopté une méthode simple pour y palier : se **détacher** de l'ordinateur et **repenser** l'implémentation du programme sur un exemple concret, au **papier**.

La compréhension du sujet aussi n'a pas été **directe**. Il nous a fallu de nombreuses lectures et questions posées aux **professeurs** afin d'être sûr d'avoir réellement bien compris les **enjeux** et les **attendus** du programme.

Une grande partie du travail qui est encore partiellement à revoir est la **gestion dynamique** de la **mémoire**. Avec autant de types de données et autant de structures de stockage, considérant le manque de GC de Ada, nous avons eu à redoubler d'efforts afin de bien libérer chaque emplacement mémoire qui aurait pu être alloué.

Finalement, nous avons aussi passé énormément de temps sur la fonction **InfixBrowsing** et les différents cas suivant l'emplacement actuel du curseur. Mais comme dit plus haut, une fois la **bonne méthode** adoptée et un **cas réel** traité, aucune fonction n'était de **taille** et nous avons pu finalement régler tous les problèmes sur le chemin.

==========

6. Organisation de l'équipe (qui a fait quoi + tableau) :

	Raffinages	ADS	Tests	ADB
Compresser	Ilian	Hocine	Hocine	Hocine/Ilian
Décompresser	Hocine			

==========

7. Bilan technique de l'avancement du projet / perspectives d'amélioration :

-- A remplir pour le 15 janvier.

==========

8. Bilan personnel et individuel ILIAN (intérêt, temps passé (général, conception, implémentation, mise au point, rapport) et enseignements tirés du projet) :

-- A remplir pour le 15 janvier.

==========

8. Bilan personnel et individuel HOCINE (intérêt, temps passé (général, conception, implémentation, mise au point, rapport) et enseignements tirés du projet) :

-- A remplir pour le 15 janvier.

A une époque où tout n'est qu'information et où la nouvelle monnaie se retrouvera bientôt être notre capacité à stocker cette dernière, la question de la compression se pose de plus en plus. Dans notre cas, nous voudrions compresser un fichier en un fichier de taille réduite, sans perdre d'information.

Notre approche se basera alors sur le principe de lossless compression (ou compression sans perte en français).
Contrairement au format .jpeg par exemple qui va réduire la taille d'une image en perdant en qualité, nous souhaitons garantir un produit 100% conforme à l'original (càd que le fichier décompressé doit être égal au fichier initial).

Le codage de Huffman se trouve alors être un parfait candidat pour notre problème : il permet de coder tous les caractères d'un fichier avec un code à longueur variable, qui permet d'utiliser un code plus court pour les caractères apparaissant souvent et plus long pour ceux qui apparaissent moins.
L'objectif de ce projet est ainsi de compresser un fichier d'entrée (en l'occurence un fichier binaire pour garantir l'universalité de la compression) en un fichier de taille réduite de 30 à 50% à l'aide du codage de Huffman et ce, sans perdre d'information.