Résumé du rapport :

Ce rapport retrace le schéma de pensée et les différentes grandes étapes de la réalisation du projet 3. La décompression d'Huffman à été implémentée avec 4 programmes différents, 2 convertisseurs binaire/texte pour faciliter son utilisation et un programme de compression et un de décompression. Ces derniers, bien que robustes et fonctionnels, sont toujours voués à évoluer. Ce fut un projet très enrichissant et nous ayant permis d'en apprendre plus sur la programmation, la gestion de projets et la recherche d'information de manière générale.

==========

Introduction du problème et contenu du rapport :

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. Ensuite à l'aide de différentes données détaillées ci-dessous, il sera possible de **décompresser** le fichier et d'obtenir le **fichier d'entrée.**

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.

Le programme de décompression d'Huffman comprend les mêmes parties que la compression :

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

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

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

==========

2. Présentation des choix réalisés :

Tout d'abord, puisque d'une part pour réaliser la compression, la manipulation de **données binaires** est la manière la plus appropriée, et, puisque d'autre part, les fichiers à compresser sont majoritairement des **fichiers textes,** nous avons décidé de créer une fonction **traduisant** un fichier texte en binaire (un caractère correspondant à son **code ASCII** en binaire) et une autre faisant l'opération réciproque. Nous pourrons aussi au final mieux **comparer** si le fichier décompressé est le même que celui d'origine.

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.

Pour ce qui est de l'algorithme de **décompression**, nous avons choisi de travailler avec les mêmes **types** que pour l'algorithme de compression pour les mêmes raisons qu' évoquées précédemment, à la différence que nous avons ajouté un nouveau type **tableau** pour stocker les caractères présents dans le fichier texte. De plus, nous n'avons pas eu le besoin d'utiliser de **table de hachage** puisque toutes les informations nécessaires à la décompression étaient accessibles **facilement** dans le fichier compressé.

==========

3. Présentation des algorithmes / types de données :

Présentation des algorithmes de compression :

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**.

Présentation des algorithmes de décompression :

Présentation de l'algorithme qui parcours le fichier compressé pour en extraire les informations importantes pour la décompression (ExploreText):

Cette procédure est divisée en 3 majeures parties.

- Récupérer le **code ASCII binaire** de chaque caractère utilisé dans le fichier texte et le stocker dans un tableau, ce tableau comporte les différents symboles dans l'ordre **infixe** de l'arbre de Huffman. Le critère d'arrêt de cette étape est le fait qu'on retrouve deux fois le **même octet**.
- Récupérer le code du parcours **infixe** de l'arbre. Cette étape s'arrête lorsqu'on a atteint le caractère **'1'** un nombre de fois égal aux nombre de **caractères stockés** dans le tableau. Ainsi, à l'issue des deux étapes il est possible de recréer l'**arbre de Huffman** et donc d'avoir accès au **codage de Huffman** de chaque caractère.
- Écrire le code ASCII de chaque caractère du fichier compressé dans le fichier décompressé jusqu'à atteindre la **fin du fichier**. On prend soin de retirer le **dernier octet** symbolisant la **fin du fichier**.

Présentation de l'algorithme recréant l'arbre de Huffman avec le parcours infixe (ReconstructHuffmanTree):

Cette procédure est une **procédure récursive**. La **condition d'arrêt** de cette procédure est lorsqu'on a un **'un'** et que **tous** les **parents** sont vus.

Si le caractère à l'indice i du code infixe est un **0** alors on crée **deux nœuds** qui seront respectivement le **fils gauche** et le **fils droit** du nœud sur lequel on a appelé la procédure. Comme le veut le parcours infixe, on rappellera la procédure sur le **fils gauche** du nœud courant.

Si le caractère à l'indice i du code infixe est un **1** il existe un **disjonction** de cas :

- Si le parent de ce nœud **n'est pas vu** alors on rappellera la procédure sur le **fils droit de ce parent** et celui-ci sera noté comme **vu**.

- Sinon tant que le nœud est **vu** on l'affecte à son parent. On rappellera la procédure sur le **fils droit de ce nœud courant** et celui-ci sera noté comme **vu**.

Cependant, dans tous les cas puisque la présence d'un **1** dans le code infixe symbolise le fait qu'on se trouve sur une **feuille** on affecte au symbole de cette feuille la valeur du **tableau des symboles** (rangé dans l'ordre **infixe**) à l'indice i. Cette valeur **i** est **incrémentée** de **1** à la suite de cette étape.

Présentation de l'algorithme qui parcourt l'arbre recrée pour obtenir le code de Huffman de chaque symbole et le traduire dans le fichier décompressé (ExploreTree):

Cette procédure est aussi une **procédure récursive** dont le **critère d'arrêt** est l'arrivée à la **fin** du fichier **compressé**.

Si la valeur du caractère du texte compressé est **0** on rappelle la procédure sur le **fils gauche**, si c'est un **1,** sur le **fils droit**. On appelle la fonction **Get** pour avancer le curseur d'un caractère dans le fichier compressé.

De plus, si le fils gauche de l'élément est **nul**, c'est-à-dire que l'on se trouve sur une **feuille**, alors on ajoute le **symbole** de cette feuille au **fichier décompressé** et on rappelle la procédure sur **la racine de l'arbre**.

==========

4. Démarche adoptée pour le test :

Tout d'abord, nous avons procédé à plusieurs tests unitaires en comparant, après chaque 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,
- **GetTextCode**, devait quant à elle renvoyer une table de hachage permettant de **reconstruire** par la suite l'arbre de Huffman lors de la décompression,
- et **ReconstructHuffmanTree** devait renvoyer le même arbre de Huffman que celui créé dans compression.

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

==========

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.

Ayant bien compris comment fonctionnait compresser nous n’avons pas éprouvé de difficultés particulières à faire décompresser.

==========

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

Organisation de l’équipe					
Raffinages		**Compresser**		**Décompresser**	
Ilian	Hocine	Ilian	Hocine	Ilian	Hocine
Compresser	Décompresser	BuildHuffmanTree	GetSymbols	ExploreText	ReconstructHuffmanTree
		EncodeText	InfixBrowsing	ExploreTree	
			DisplayTree		

==========

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

Les programmes écrits effectuent les tâches voulues en un temps réduit et sans erreurs. Le cahier des charges est alors respecté et le produit est donc parfaitement au normes. Cependant, certains éléments restent tout de même à améliorer par la suite afin de garantir une expérience la meilleure possible :

→ Permettre de n'avoir aucune fuite de mémoire, bien que ces dernières ont été traitées durant le développement.

→ Afficher l'arbre de Huffman lors de l'activation du mode bavard de la compression sans les barres en dessous des 1 qui peuvent porter à confusion l'utilisateur.

→ Changer d'algorithme de tri et en adopter un en complexité réduite (passer de $O(n^2)$ en $O(n)$ par exemple).

==========

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

Ce projet m'a permis de renforcer mes connaissances en ADA que ce soit sur la manipulation des modules ou sur la manipulation des fichiers. De plus, j'ai pu découvrir le travail en équipe pour un projet de programmation et l'organisation qui en découle. J'ai apprécié cet aspect puisque ceci est la norme dans le monde professionnel. Par ailleurs, j'ai aussi apprécié le fait de réaliser un projet long s'étalant sur plusieurs semaines puisque cela requiert d'avoir une bonne organisation avec son binôme pour mener à bien les différentes étapes du projet. Pour la totalité du projet (raffinages, implémentations et fichiers annexes), j'ai passé environ une soixantaine d'heures.

==========

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

De par ce projet, j'ai pu tout d'abord avoir une meilleure compréhension des structures de données existantes et de la programmation orientée objet. Ce fut l'occasion pour moi de voir l'importance d'avoir un plan structuré lorsque l'on s'attaque à un nouveau projet, et surtout de la communication dans une équipe. Avoir l'opportunité de travailler sur un projet de la sorte permet aussi d'être plus conscient du code que l'on écrit : il faut que le reste de l'équipe puisse comprendre aisément ce que nous avons voulu faire, mais aussi, il va de soit que sur des programmes aussi longs, les redondances et autres stigmates du code spaghetti doivent être remplacées au fur et à mesure afin de garantir la pérennité du programme. Sur son entièreté, le programme m'a pris une soixantaine d'heures entre la conception, l'écriture des raffinages, le débogage et les heures passées à expérimenter sur différents choix de structures / procédures à utiliser.