École Centrale de Lille

AAP - Algorithmique Avancée et Programmation

Fil Rouge 2022

Groupe D:

BASTOS SANTOS Vitória
FERREIRA BRASIL Rebeca
LESSA GUERRA Bernardo
REIS GUEVARA David

1. Introduction

Le développement de ce travail a été divisé en trois parties et son objectif est de créer un programme qui doit pouvoir compresser et décompresser un texte en utilisant la codage de Huffmann.

Le codage de Huffman a été développé par David A. Huffman en 1952, un étudiant au MIT. C'est un algorithme de compression de données qui utilise un code de préfixe pour représenter un ensemble de symboles.

Le fonctionnement dépend d'une méthode de compression des données en attribuant des mots de code plus courts aux symboles qui apparaissent plus fréquemment dans un ensemble de données, et des mots de code plus longs aux symboles qui apparaissent moins fréquemment. Cela permet un codage et un décodage plus efficaces des données, car les mots de code plus courts occupent moins d'espace et nécessitent moins de bits à transmettre.

Afin de développer un code de Huffman, il est d'abord nécessaire de créer un tableau de fréquences de tous les symboles du jeu de données. Cette partie du programme est expliquée dans la section 2.1 du Compte Rendu. Ensuite, les symboles sont disposés dans un arbre binaire, avec les symboles les moins fréquents en bas et les plus fréquents en haut. Dans le cas de ce Fil Rouge, les symboles sont des caractères. La définition des mots de code pour chaque symbole est effectuée en suivant le chemin de la racine de l'arbre au nœud feuille du symbole, où 0 représente un virage à gauche et 1 représente un virage à droite.

Le codage Huffman est utilisé à la fois dans les applications de compression de données telles que les fichiers image et audio, mais il est également utilisé dans les systèmes de communication pour une transmission efficace des données. C'est fondamentalement un algorithme de compression sans perte, sans aucune information perdue pendant le processus de compression et de décompression.

En bref, le codage de Huffman est un algorithme de compression de données puissant mais simple utilisé dans une grande variété d'applications. Son efficacité et sa capacité à compresser efficacement les données tout en préservant l'intégrité des données d'origine font sa renommée.

2. Développement

2.1. Programme 1: heapsort.exe

2.1.1. test_utils.h et test_utils.c

- ***testutils.h***

Ce code est un fichier d'en-tête qui définit les types et les fonctions liés aux algorithmes de tri. Il commence par inclure le fichier d'en-tête "time.h", qui déclare les fonctions et les types pour gérer les informations en C.

Ensuite, c'est défini un certain nombre de typedefs et d'autres éléments. "RaZ" est une forme abrégée de "memset", qui est une fonction de la bibliothèque standard C qui remplit un bloc de mémoire avec une valeur spécifiée. "MAX_ELT" définit le nombre maximum d'éléments pouvant être traités par les fonctions de tri. Le typedef "T_elt" définit

un type pour stocker des éléments individuels des données d'entrée. Le typedef "T_data" définit un type pour stocker un pointeur vers un tableau d'éléments et un seul élément.

Ensuite, le typedef "T_stat" définit un type pour stocker des statistiques sur les performances d'une fonction de tri, y compris le nombre de comparaisons et d'opérations effectuées et la durée de la fonction en tops d'horloge. Un autre typedef, "T_pFToTest", définit un type pour un pointeur vers une fonction qui prend une variable "T_data" et un entier comme arguments et renvoie une valeur "T_data". "T_fnToTest" définit après un type pour stocker le nom, le pointeur de fonction et la taille maximale d'une fonction de tri.

C'est déclaré aussi un certain nombre de variables globales et de constantes. La variable "stats" est une variable "T_stat" qui stocke des statistiques sur les performances d'une fonction de tri. Le tableau "nb" est un tableau d'entiers qui stocke le nombre d'éléments à trier pour chaque test. La variable "outputPath" est une chaîne qui spécifie le chemin des fichiers de sortie. Les constantes "MODE_TAB_ORDONNE", "MODE_TAB_ALEATOIRE", "MODE_TAB_INVERSE", et "MODE_EVAL_X" sont des entiers qui représentent différents modes d'initialisation des données d'entrée. Le typedef "T_mode" définit un type pour stocker un mode et d'autres informations liées à l'initialisation des données d'entrée.

"testutils.h" déclare ensuite un certain nombre de fonctions pour initialiser et afficher les données d'entrée, créer des fichiers de sortie, tester les fonctions de tri et générer des données pour les fonctions de tri. La fonction "Initialiser" initialise un tableau de variables "T_elt" avec un mode spécifié. La fonction "estOrdonne" vérifie si un tableau de variables "T_elt" est ordonné d'une manière spécifiée. La fonction "Afficher" affiche le contenu d'un tableau de variables "T_elt". La fonction "Creer_Fichiers" crée des fichiers de sortie pour tracer les résultats d'une fonction de tri. Les fonctions "Test_Fn" et "Test_FnV2" testent une fonction de tri avec un nombre maximal d'éléments et un mode spécifiés. Après cela, on lit la fonction "genData" et ferme le fichier.

```c
#include <time.h>

// NB : code extrait des sujets produits par Christian Vercauter à l'occasion des TPs de CAA

#ifndef _SORT_H_
  #define _SORT_H_

#define RaZ(r) memset(&(r), 0, sizeof(r))
#define MAX_ELT                 250000

typedef int T_elt;

typedef struct {
  T_elt elt;
  T_elt * pElt;
} T_data;

typedef struct {
    unsigned long long  nbComparisons; // nombre de comparaisons effectuées
    unsigned long long  nbOperations;  // nombre d'opérations (affectations, multiplications suivant les problèmes)
    clock_t             duration_clock_t; // durée de la fonction en nombre de tops d'horloge
} T_stat;

// Type pointeur sur une fonction
// Cette fonction prend de manière générale deux arguments :
// * un pointeur sur un T_Elt
// (peut être l'adresse de début d'un tableau, ou simplement l'adresse d'une valeur à utiliser dans la fonction)
// * un entier relatif à la taille du problème considéré

typedef T_data (*T_pFToTest) ( T_data d, int n);

// Structure définissant une fonction à tester
typedef struct {
    char *name; //nom de la fonction
    T_pFToTest fn; //pointeur sur la fonction a tester
    unsigned int limit; // nombre maximal d'élements qu'elle peut manipuler dans un temps raisonnable
} T_fnToTest;

extern T_stat stats;
```

```c

#define MODE_TAB_ORDONNE 0
#define MODE_TAB_ALEATOIRE 1
#define MODE_TAB_INVERSE 2
#define MODE_EVAL_X 3

typedef struct {
  int mode;     // parmi les constantes précédentes
  char * label;
  T_elt x;    // si MODE_EVAL_X : besoin d'une variable pour exécuter la fonction, quelle valeur utiliser ?
  int checkOrder; // faut-il vérifier si la table est ordonnée ?
} T_mode;

extern int nb[];
extern char * outputPath;

// Nombre de valeurs affichées par ligne
#define NB_PER_LINE            10

// Longueur maximale de la ligne de commande construite pour appeler GnuPlot
#define CMDLINE_MAX            64

// Définition du nom de l'apllication GnuPlot
#define GNUPLOT                "gnuplot"

void Initialiser (T_elt T[], int n, int mode);
int estOrdonne (T_elt T[], int n, int strict);
void Afficher(T_elt T[], int n);

void Creer_Fichiers(const char *nom, char *Filename_log, char *Filename_plt, T_mode m);
void Test_Fn ( const char * nom, T_pFToTest fn , T_elt Table[], int nmax, T_mode m);
void Test_FnV2 ( const char * nom, T_pFToTest fn , T_elt Table[], int nmax, T_mode m);
T_data genData(T_elt e, T_elt * pE);

#endif
```

- ***test_utils.c***

Le code élabore les définitions faites dans test_utils.h pour créer les tables et leur application dans le cadre de la résolution de la question 1, établissant les fonctions "genData" (de type T_data), "Initialiser" (void), "estOrdonne" (int) , "Afficher", "Creer_Fichiers", "Test_Fn" et "Test_FnV2".

2.1.2. elt.h et elt.c

- Utilisés pour définir de type T_Elt avec les fonctions `T_elt genElt ()` et `char * toString()`

2.1.3. heapsort.h

Ce code est un fichier d'en-tête pour une implémentation de l'algorithme de tri en C. Le fichier tête fournit les signatures de fonction pour un certain nombre de fonctions qui seront définies dans le fichier source heapsort.c correspondant.

Le fichier d'en-tête commence par inclure deux autres fichiers d'en-tête: "elt.h" et "test_utils.h". Ces fichiers d'en-tête contiennent des définitions et des déclarations requises par l'implémentation du tri. Le heapsort.h définit ensuite un certain nombre de directives de préprocesseur qui permettent au programmeur de définir des constantes ou d'effectuer une simple substitution de texte dans le code source. Dans ce cas, "MAXHEAP" ou "MINHEAP" sera définie, selon celle qui n'est pas commentée. Ces éléments contrôlent le comportement de l'algorithme de tri d'une manière ou d'une autre, par exemple s'il trie par ordre croissant ou décroissant.

Le fichier d'en-tête déclare alors la fonction "heapSort_Aux", qui prend deux arguments : une variable de type "T_data" et un entier. La fonction renvoie une valeur de type "T_data". La fonction fait la première partie du problème, c'est-à-dire trier l'arbre. C'est également déclaré une fonction "makeheap", qui prend trois arguments : un tableau de variables "T_elt", un entier et un autre entier.

La fonction "swap", qui prend deux pointeurs vers des variables "T_elt", échange les valeurs vers lesquelles ils pointent. Il déclare également une fonction "printArray", qui prend un tableau de variables "T_elt" et un entier, et imprime le contenu du tableau.

Enfin, le fichier se ferme avec une garde d'inclusion, qui est une directive de préprocesseur qui empêche le contenu du fichier d'en-tête d'être inclus plus d'une fois dans un seul fichier source.

```
T_data heapSort_Aux(T_data d, int n);
T_data heapSort(T_data d, int n);
void makeheap(T_elt arr[], int n, int i);

void swap(T_elt *a, T_elt *b);
void printArray(T_elt arr[], int n);
```

2.1.4. heapsort.c

Ce code est une implémentation de l'algorithme de tri par tas en C. L'algorithme de tri par tas est un algorithme de tri qui fonctionne en construisant d'abord un tas (un arbre binaire complet où la valeur de chaque nœud parent est supérieure ou égale à ses nœuds enfants) à partir des données d'entrée, puis en triant les données en supprimant à plusieurs reprises le nœud racine du tas (qui est le plus grand élément dans le cas d'un MAXHEAP ou le plus petit élément dans le cas d'un MINHEAP) en l'ajoutant à la liste triée et en ajustant le tas restant pour conserver la propriété du tas.

Au début du code, nous incluons les fichiers d'en-tête de la bibliothèque C standard "stdio.h" et "string.h", ainsi que deux fichiers d'en-tête supplémentaires "traces.h" et "heapsort.h". "traces.h" et "heapsort.h" contiennent des définitions et des déclarations requises par l'implémentation du tri par tas.

C'est définit alors un certain nombre de fonctions. La fonction "swap" prend deux pointeurs vers des variables "T_elt" et échange les valeurs vers lesquelles ils pointent. La fonction "printArray" prend un tableau de variables "T_elt" et un entier, et imprime le contenu du tableau. La fonction "heapSort_Aux" prend deux arguments : une variable de type "T_data" et un entier. La fonction renvoie une valeur de type "T_data".

La fonction "heapSort" est la fonction principale pour effectuer l'algorithme de tri par tas. Elle prend deux arguments: une variable de type "T_data" et un entier et renvoie une valeur de type "T_data". La fonction appelle d'abord la fonction "heapSort_Aux" (fonction auxiliaire) avec les données d'entrée et la taille. Il crée ensuite un tableau "Finale" et y copie les données triées du tableau "Aux1". Enfin, il appelle à nouveau la fonction "heapSort_Aux", en transmettant une partie du tableau "Aux1" et la taille restante du tableau, et renvoie les données triées résultantes dans le tableau "Finale".

Le code définit ensuite deux versions de la fonction "makeheap", une pour construire un MAXHEAP et une pour construire un MINHEAP. Ces fonctions prennent trois arguments : un tableau de variables "T_elt", un entier représentant la taille du tas et un entier représentant l'indice du nœud racine du sous-arbre à entasser. La fonction ajuste les valeurs

dans le tableau d'entrée pour conserver la propriété de tas. La version de la fonction "makeheap" qui est utilisée dépend de laquelle des macros "MAXHEAP" ou "MINHEAP" est définie au début du code.

- **Fonction *heapSort_Aux***

```
T_data heapSort_Aux(T_data d, int n){
    T_elt *arr = d.pElt;
    int lastNonLeafNode = (n / 2)-1 ;
    for (int i = lastNonLeafNode; i >= 0; i--){
        ++stats.nbComparisons;
        makeheap(arr, n, i);
    }
    return genData(0, arr);
}
```

- **Fonction *heapSort***

```
37 ▼ T_data heapSort(T_data d, int n){
38     //Here n = Number of elements +1
39       T_data Aux2 = heapSort_Aux(d, n);
40       T_elt *Aux1 = Aux2.pElt;
41       T_elt *Finale;
42       Finale = Aux1;
43
44 ▼     for (int i=0; i<n; i++) {
45           Finale[i] = Aux1[i];
46           heapSort_Aux(genData(0, &Aux1[i]), n-(i));
47      }
48       return genData(0, Finale);
49  }
```

- **Fonction make*heap***

Pour MAXHEAP:

```
void makeheap(T_elt arr[], int n, int i){

int biggest = i; // Initialize largest as root
int leftChild = 2 * i + 1; // left child = 2*i + 1
int rightChild = 2 * i + 2; // right child = 2*i + 2

// If left child is greater than root
    if (leftChild < n && arr[leftChild] > arr[biggest]){
        biggest = leftChild;
        ++stats.nbComparisons;
        ++stats.nbOperations;
    }
    if (rightChild < n && arr[rightChild] > arr[biggest]){
        biggest = rightChild;
        ++stats.nbComparisons;
        ++stats.nbOperations;
    }

// If largest is not the root
    if (biggest != i){
        // swap root with the new largest
        swap(&arr[i], &arr[biggest]);
        makeheap(arr, n, biggest);
    }
}
```

Pour MINHEAP:

```
void makeheap(T_elt arr[], int n, int i){

int largest = i; // Initialize largest as root
int leftChild = 2 * i + 1; // left child = 2*i + 1
int rightChild = 2 * i + 2; // right child = 2*i + 2

// If left child is greater than root

    if (leftChild < n && arr[leftChild] < arr[largest]){
        largest = leftChild;
        ++stats.nbComparisons;
        ++stats.nbOperations;
    }

// If right child is greater than new largest

    if (rightChild < n && arr[rightChild] < arr[largest]){
        largest = rightChild;
        ++stats.nbComparisons;
        ++stats.nbOperations;
    }

// If largest is not the root

if (largest != i){
// swap root with the new largest
    swap(&arr[i], &arr[largest]);
    makeheap(arr, n, largest);
}
}
```

- ***main.c***

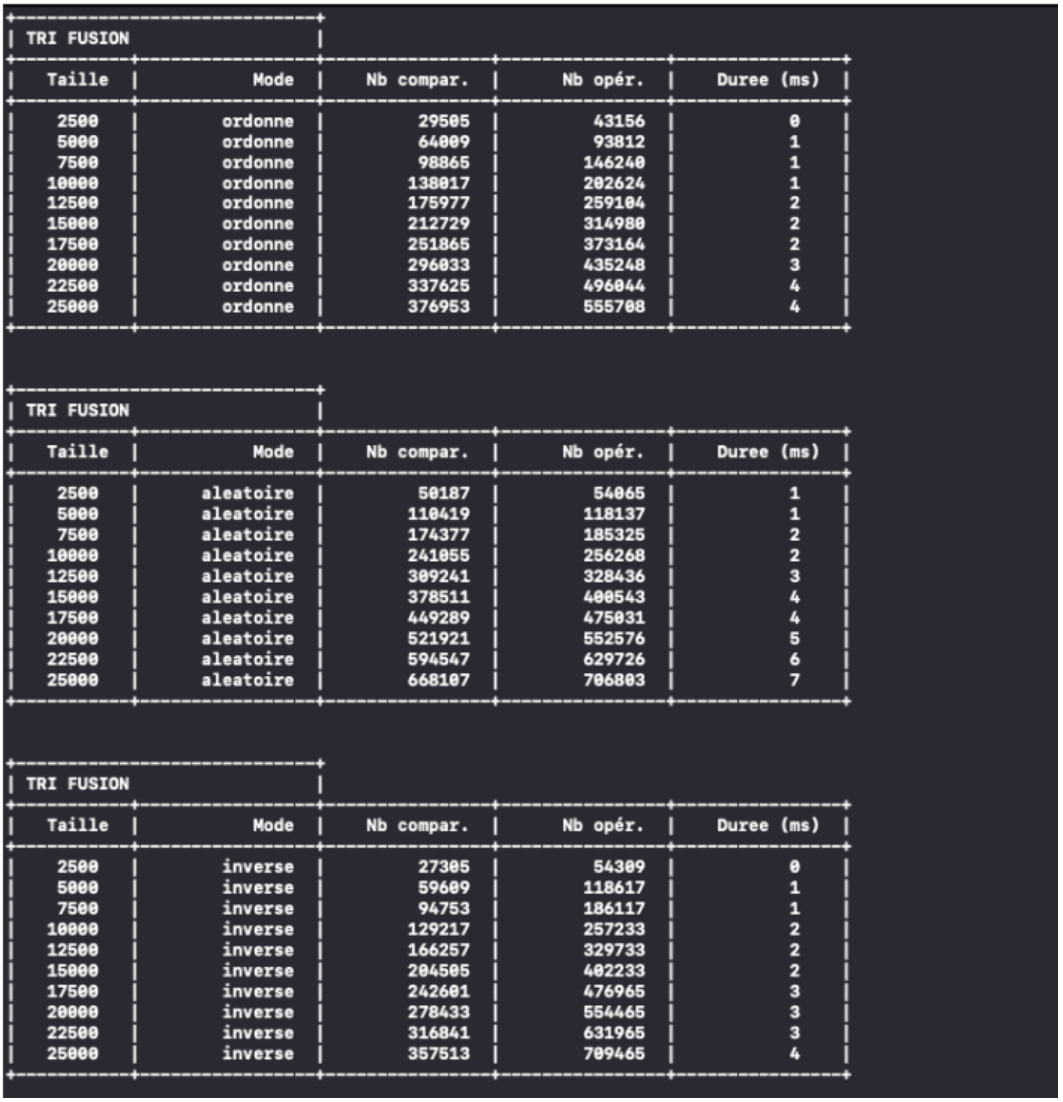
C'est le fichier avec le code principal du programme 1, qui teste la fonction heapSort, qui trie un tableau d'éléments. Le programme au début comprend test_utils.h.

Dans la fonction principale, c'est créé un tableau et elle initialise un générateur de nombres pseudo-aléatoire. Ensuite, la fonction Test_Fn est appelée plusieurs fois, en lui transmettant à chaque fois la fonction heapSort et un onglet de tableau d'éléments MAX_ELT, ainsi qu'un élément différent du tableau m chaque fois (m [MODE_TAB_ORDONNE], m[MODE_TAB_ALEATOIRE] et m[MODE_TAB_ALEATOIRE]). La fonction Test_FnV2 est appelée ensuite quelques fois, de la même façon que la fonction Test_Fn, sauf avec une valeur fixe de 512 pour le deuxième argument. Le bloc de code commenté à la fin de la fonction principale sert à générer une structure de données appelée d en utilisant la fonction genData, la trier en utilisant à la fois heapSort_Aux et heapSort, puis imprimer le tableau trié résultant.

```c
#include <stdio.h>
#include <stdlib.h>

#include "test_utils.h"

// Ajouter ici les prototypes des fonctions à tester
//T_data fusionSort(T_data d, int n);
T_data heapSort(T_data d, int n);
//void fusionsort(void *base, size_t nmemb, size_t size, int (*compar)(const void *, const void *));

// mode, label, x, checkOrder
T_mode m[] = {
  {MODE_TAB_ORDONNE, "ordonne", 0, 1},
  {MODE_TAB_ALEATOIRE, "aleatoire", 0, 1},
  {MODE_TAB_INVERSE, "inverse", 0, 1},
  {MODE_EVAL_X, "x=2.0", 2.0, 0},
  {MODE_TAB_ORDONNE, "ordonne (x=59)", 59, 0},
  {MODE_TAB_ORDONNE, "hanoi", 1, 0}
};

int main(int argc, char *argv[])
{
    T_elt tab [MAX_ELT];
  //T_data d;
  //int base[MAX_ELT];

  outputPath = "output"; // indiquer le chemin du répertoire où créer les fichiers
  // NB: s'il est relatif, il sera relatif au répertoire depuis lequel l'exercice est exécuté

    // Initialisation du générateur de nombres pseudo-aléatoires
    srand((unsigned int)time(NULL));

  Test_Fn("TRI FUSION", heapSort, tab, MAX_ELT/10, m[MODE_TAB_ORDONNE] );
  Test_Fn("TRI FUSION", heapSort, tab, MAX_ELT/10, m[MODE_TAB_ALEATOIRE] );
  Test_Fn("TRI FUSION", heapSort, tab, MAX_ELT/10, m[MODE_TAB_INVERSE] );

  Test_FnV2("TRI FUSION", heapSort, tab, 512, m[MODE_TAB_ORDONNE] );
  Test_FnV2("TRI FUSION", heapSort, tab, 512, m[MODE_TAB_ALEATOIRE] );
  Test_FnV2("TRI FUSION", heapSort, tab, 512, m[MODE_TAB_INVERSE] );



 /*
    T_data d;
    T_elt v[13]={5,7,2,12,11,1,13,22,4,9,17,12,10};
    d=genData(0, v);


    printArray(v, 13);
    T_data out0 = heapSort_Aux(d, 13);
    printArray(out0.pElt, 13);

    T_data out = heapSort(d, 13);
    printArray(out.pElt, 13);
 */
}
```

2.1.5. Comparaison avec TEA3

Comme on peut le voir ci-dessous, les résultats Tri Fusion pour TEA3 avaient une durée plus courte que les résultats pour la première partie du fil rouge.
Les résultats du TEA3 étaient:



TRI FUSION

Taille	Mode	Nb compar.	Nb opér.	Duree (ms)
2500	ordonne	29505	43156	0
5000	ordonne	64009	93812	1
7500	ordonne	98865	146240	1
10000	ordonne	138017	202624	1
12500	ordonne	175977	259104	2
15000	ordonne	212729	314980	2
17500	ordonne	251865	373164	2
20000	ordonne	296033	435248	3
22500	ordonne	337625	496044	4
25000	ordonne	376953	555708	4

TRI FUSION

Taille	Mode	Nb compar.	Nb opér.	Duree (ms)
2500	aleatoire	50187	54065	1
5000	aleatoire	110419	118137	1
7500	aleatoire	174377	185325	2
10000	aleatoire	241055	256268	2
12500	aleatoire	309241	328436	3
15000	aleatoire	378511	400543	4
17500	aleatoire	449289	475031	4
20000	aleatoire	521921	552576	5
22500	aleatoire	594547	629726	6
25000	aleatoire	668107	706803	7

TRI FUSION

Taille	Mode	Nb compar.	Nb opér.	Duree (ms)
2500	inverse	27305	54309	0
5000	inverse	59609	118617	1
7500	inverse	94753	186117	1
10000	inverse	129217	257233	2
12500	inverse	166257	329733	2
15000	inverse	204505	402233	2
17500	inverse	242601	476965	3
20000	inverse	278433	554465	3
22500	inverse	316841	631965	3
25000	inverse	357513	709465	4

Les résultats du heapsort.exe sont:

HEAPSORT				
Taille	Mode	Nb compar.	Nb opér.	Duree (ms)
2500	ordonne	1563750	0	3
5000	ordonne	6252500	0	12
7500	ordonne	14066250	0	26
10000	ordonne	25005000	0	49
12500	ordonne	39068750	0	72
15000	ordonne	56257500	0	102
17500	ordonne	76571250	0	135
20000	ordonne	100010000	0	202
22500	ordonne	126573750	0	324
25000	ordonne	156262500	0	414

HEAPSORT				
Taille	Mode	Nb compar.	Nb opér.	Duree (ms)
2500	aleatoire	1741463	354762	6
5000	aleatoire	6992316	1478320	27
7500	aleatoire	15648071	3161654	56
10000	aleatoire	27847152	5681711	98
12500	aleatoire	43438495	8736146	161
15000	aleatoire	62700251	12881609	248
17500	aleatoire	85185254	17223440	315
20000	aleatoire	111320381	22615518	467
22500	aleatoire	140675625	28197942	568
25000	aleatoire	173807945	35084342	651

HEAPSORT				
Taille	Mode	Nb compar.	Nb opér.	Duree (ms)
2500	aleatoire	1745794	363418	7
5000	aleatoire	6957779	1409240	25
7500	aleatoire	15632660	3130886	61
10000	aleatoire	27845956	5679241	107
12500	aleatoire	43444076	8747360	166
15000	aleatoire	62615898	12712880	230
17500	aleatoire	85058168	16969201	286
20000	aleatoire	111037470	22049744	440
22500	aleatoire	140797927	28442554	536
25000	aleatoire	173811009	35090491	606

2.2. Programme 2: codage.exe

Le groupe a développé le programme "coding.exe" qui prend un texte comme entrée standard et applique le codage Huffman pour compresser les caractères en codage binaire. Le programme affiche ensuite la table de codage résultante, le texte compressé et un bilan indiquant le taux de compression sur sa sortie standard.

Le code est conçu pour être canonique, ce qui signifie qu'il suit l'ordre lexicographique des caractères lors de leur extraction du minimiseur indirect et de la création de l'arbre de codage. Cela garantit que l'arbre de codage est optimisé en fonction du nombre d'occurrences de chaque caractère dans le texte.

2.2.1. heapsort_V2.h

Le fichier d'en-tête définit le nouveau type qui est adapté pour faire le procedure similaire de la fonction heapsort de la partie 1.

```
37 ▼ typedef struct {
38       unsigned int nbElt; //number effective of the nods
39       unsigned char tree[MAXCARS]; //organization of the nods
40       int data[2*MAXCARS-1]; // table that contains the occurences
41   } T_indirectHeap;
```

2.2.2. heapsort_V2.c

Les fonctions qui doivent être expliqués maintenant en comparaison avec le heapsort.c sont les fonctions *searchInt* et *search*. Ces fonctions, à partir d'un vecteur, prennent un caractère ou un entier (selon la fonction utilisée), cherchent où il se trouve dans le vecteur et renvoient l'indice de sa position.

```
85 ▼ int searchInt(int *vector, int a){
86       int j=0;
87 ▼     for(int i=0; i<MAXCARS; i++){
88           j++;
89 ▼         if (vector[i]==a) {
90               i=MAXCARS-1;
91           }
92       }
93       return j;
94   }
```

```
96 ▼ int search(char *vector, char a){
97       int j=0;
98 ▼     for(int i=0; i<MAXCARS; i++){
99           j++;
100 ▼        if (vector[i]==a) {
101              i=MAXCARS-1;
102          }
103      }
104      return j;
105  }
```

Un sujet important dans ce code est qu'on a addapté les fonctions maximier et minimier au nouveau type que nous utilisons dans le code.

2.2.3. codage.h

Le fichier d'en-tête définit les types et les fonctions liés aux algorithmes du codage.c, comme T_box.

```
16 ▼ typedef struct{
17       int bin[7];
18       char letre;
19       int freq;
20   }T_box;
```

2.2.4. codage.c

Ce code explicite quelques fonctions importantes pour l'objectif général du programme 2:

- *counting*

Fonction qui compte la fréquence de chaque caractère et renvoie T_data

```
T_indirectHeap counting(T_indirectHeap t, char *c) { // give us the vector initial of occurences (pré data)
    int size = strlen(c);
    for(int i=0;i<2*MAXCARS-1;i++){
        t.data[i]=0;
    }

    for (int i=0;i<size;i++) {
        t.data[c[i]]++;

    }
  return t;
}
```

- *initTree*

La fonction voit où il n'y a pas de zéro et l'ajoute à l'arbre, et comme ça, l'arbre est initialisée

```
// initialize the tree
T_indirectHeap initTree(T_indirectHeap t) {
  int j = 0;
  for (int i = 0; i < 2 * MAXCARS - 1; i++) {
    if (t.data[i] != 0) {
      t.tree[j++] = i;
      //printf("%i \n", i);
    }
  }
  return t;
}
```

- *numElt0*

La fonction renvoie le nombre d'éléments utiles qui existent réellement

```
T_indirectHeap numElt0(T_indirectHeap t) {
  int nb_Elt=0;
  for (int i = 0; i < 2 * MAXCARS - 1; i++) {

    if (t.data[i] != 0) {
      nb_Elt++;
    }
  }
  t.nbElt = nb_Elt;
  return t;
}
```

- *initHeap0*

La fonction est responsable pour initialiser l'indirectHesp

```
T_indirectHeap *initHeap0(T_indirectHeap t, char *c) {
    T_indirectHeap *pAux=malloc(sizeof(T_indirectHeap));
    pAux = &t;
    for (int i = 0; i < 2 * MAXCARS - 1; i++) {
    t.data[i] = 0;
  }
  t = counting(t, c);
  t = initTree(counting(t, c));
  t = numElt0(counting(t, c));
    return pAux;
}
```

- *initHuffmanTree*

Cela sert à mettre dans tous les espaces -256

```
int *initHuffmanTree(void) {
    int *p;
    p = malloc(256*sizeof(int));
    for (int i = 0; i < 2 * MAXCARS - 1; i++) {
    p[i] = -256;

  }
  for(int i=0;i<2 * MAXCARS - 1; i++){
    if(p[i] != -256){
      printf("Valor incorreto inicializado");
      }
  }
    return p;
}
```

- *insertdata*

La fonction est responsable pour réinsérer les résultats du tri dans data

```
void insertData(T_indirectHeap *p, int a, int b, int i){
   int result = p->data[a] + p->data[b];
   p->data[(128+i)] = result;
}
```

- *freq*

La fonction a le même rôle que la fonction "counting", mais avec des paramètres d'entrée différents.

```
int *freq(char *texto, int *occ){//ok
    //int occ[MAXCARS];//={0};
    int size = strlen(texto);
    for(int k=0;k<MAXCARS;k++){
        occ[k]=0;
    }
    for (int i=0;i<size;i++) {
        occ[texto[i]]++;
    }
    return occ;
}
```

- *nbElt*

La fonction a le même rôle que la fonction "numElt0", mais avec des paramètres d'entrée différents.

```
int nbElt(int *occ){//ok
    int nbElt=0;
    for (int k=0;k<MAXCARS;k++) {
        if(occ[k]!=0){
            nbElt++;
        }
    }
    return nbElt;
}
```

- *initBoxes*

Cette fonction initialise un tableau de structures T_box et renvoie un pointeur vers le premier élément du tableau. La structure T_box a trois membres : char letre, int freq et int bin[7]. Il alloue de la mémoire pour un tableau de structures T_box qui est de la taille de nbElt(occ) multiplié par la taille d'une seule structure T_box. La mémoire de ce tableau est alors affectée à pBox.

```
147 ▼ T_box *initBoxes(T_box v[], int *occ){//ok
148       T_box *pBox=v;
149       pBox=malloc(nbElt(occ)*sizeof(T_box));
150       int l=0;
151 ▼     for(int i=0;i<MAXCARS;i++){
152 ▼         if(occ[i]!=0){
153               pBox[l].letre = i;
154               pBox[l].freq = occ[i];
155 ▼             for(int k=0;k<7;k++){
156                   pBox[l].bin[k]=-1;
157               }
158               l++;
159               //pBox++;
160           }
161       }
162       return pBox;
163   }
```

- *huffman*

La fonction entre dans une boucle qui continue jusqu'à ce que l'élément nbElt de Mi soit supérieur à 1. À l'intérieur de la boucle, elle déclare une variable f et la définit égale à 128 + i. Il appelle ensuite la fonction removeMax sur Mi et stocke le résultat dans une structure T_elt appelée C1. Il appelle ensuite la fonction buildHeap sur Mi et appelle à nouveau la fonction removeMax sur Mi, stockant le résultat dans une structure T_elt appelée C2. Ensuite, il insère en conséquence les éléments sur Ht et le renvoie.

```
212 ▼ int *huffman(char *texto){
213
214       T_indirectHeap t;
215       T_indirectHeap *Mi=NULL;
216
217       Mi = initHeap0(t,texto); // Initializes the initial minimier
218       int const ntotale = Mi->nbElt; //Needed for the loop
219       int *Ht=initHuffmanTree();
220
221
222       buildHeap(Mi); // Réorganisation du minimier
223       generatePNG("Dot_Inicial", Mi, Ht);
224
225       int i=0;
226 ▼     while(Mi->nbElt>1){
227           unsigned char f = 128 + i;
228           T_elt C1 = removeMax(Mi); // Extraire et réorganiser
229           buildHeap(Mi);
230
231           T_elt C2 = removeMax(Mi); // Extraire et réorganiser
232           buildHeap(Mi);
233
234           insertHuffTree(Ht, C1, C2, i);// Ajout dans l'arbre de codage
235           insertData(Mi, C1, C2, i); // Ajout dans tableau de frequences
236           addElt(Mi, f); //Add the new element in the tree
237           buildHeap(Mi);
238           i++;
239       }
240           generatePNG("Dot_Final", Mi, Ht);
241            return Ht;
242
243   }
```

- *Codage*

La fonction est responsable pour parcourir l'arbre de Huffman.

```c
void codage(T_box *v, int Ht[], int nbElt){
    int indiceHt =v->letre;
    int indiceBin;

    for(indiceBin=0;indiceBin<7;indiceBin++){
        //printf(" %d %d %d\n ", indiceBin, indiceHt, Ht[indiceHt]);
        if(Ht[indiceHt]==126+nbElt){
            v->bin[indiceBin]=1;
            break;
        }
        if(Ht[indiceHt]==(-1)*(126+nbElt)){
            v->bin[indiceBin]=0;
            break;
        }
        else{
            if(Ht[indiceHt]>0){
                v->bin[indiceBin]=1;
                indiceHt=Ht[indiceHt];
            }
            else{
                v->bin[indiceBin]=0;
                indiceHt=-Ht[indiceHt];
            }
        }


    }


}
```

2.2.5. main.c

La fonction main initialise d'abord certaines variables et efface l'écran. Ensuite, c'est obtenu le texte à compresser et la fréquence de chaque caractère dans le texte est calculé. Elle initialise un vecteur de boîtes contenant des informations sur chaque caractère et génère l'arbre de Huffman pour le texte.

Ensuite, elle ouvre un fichier pour la sortie et écrit le caractère, la fréquence et le code Huffman pour chaque caractère du texte dans le fichier. Elle convertit ensuite le texte en sa représentation en code Huffman, calcule les tailles du texte d'origine et de sa représentation en code Huffman, et affiche le taux de compression des données résultant. Cela est fait avec le type qui sera utilisé au programme 3. Enfin, il ferme le fichier de sortie et le programme se termine.

```c
int main(int argc, char ** argv) {

    CLRSCR();
    WHOAMI();
    FILE *output;
    char c=0;
    char *entrada;
    int *Tout;
    int *Ht=NULL; //The huffman tree

    printf("Inicializando Heap \n");
    entrada = getText(c); //Obtain the text to compress and stores in entrada

    int *occ=malloc(MAXCARS*sizeof(int));
    occ = freq(entrada, occ);//Recreates the occurences vector

    int nbelt=nbElt(occ); //Number of diferent characters in the entry
    int size_string = nbChar(occ); //Number of characters in the entry text
    T_box *pcaixa; //The vector containing the result of the compression

    pcaixa=initBoxes(pcaixa, occ);
    Ht=huffman(entrada);

    output = fopen("saidas.txt", "w");
    if(output == NULL){
      printf("Erro na abertura do arquivo!");
    }

    for(int l=0; l<nbelt;l++){
        printf("\n");
        codage(&pcaixa[l], Ht, nbElt(occ));
        printf("\n | %c | %d | ", pcaixa[l].letre, pcaixa[l].freq);
        fprintf(output, "%c, %d,", pcaixa[l].letre, pcaixa[l].freq);
        for(int k=6;k>-1;k--){
            if(pcaixa[l].bin[k]!=-1){
                printf("%d ", pcaixa[l].bin[k]);
                fprintf(output,"%d", pcaixa[l].bin[k]);
            }
        }
        fprintf(output,"\n");
        printf("|\n");

    }
    fprintf(output,"END OF FILE,\n");
```

2.2.6. Output

```
FilRouge_pt2 (Jan  4 2023 00:45:59)

Inicializando Heap
ABBACADABRA


 | A | 5 | 1 |


 | B | 3 | 0 1 |


 | C | 1 | 0 0 1 |


 | D | 1 | 0 0 0 0 |


 | R | 1 | 0 0 0 1 |
Le code de Huffman est: 10101100110000010100011
Longueur du code binaire : 33 bits
Longueur du code de huffman : 22 bits
Ratio de compression : 66.67
Process exited with status 0
logout

Saving session...
...copying shared history...
...saving history...truncating history files...
...completed.

[Processo concluído]
```

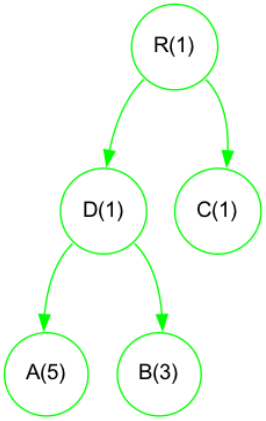
Fichier de texte géneré:

saidas.txt

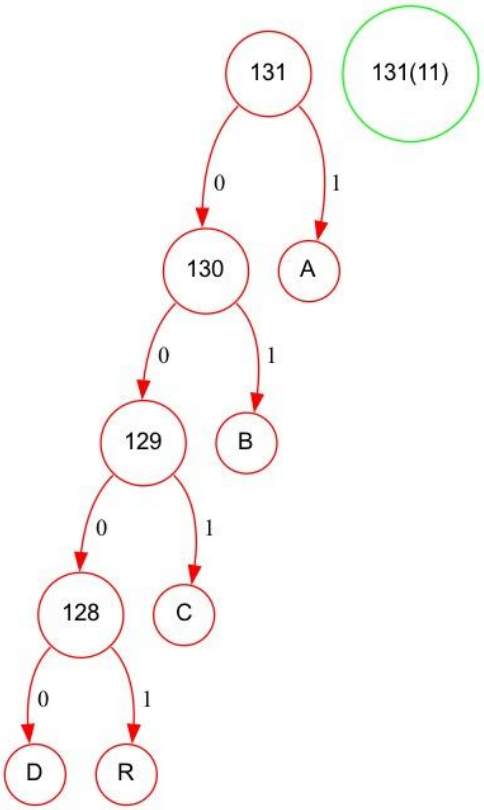
```
A, 5,1
B, 3,01
C, 1,001
D, 1,0000
R, 1,0001
END OF FILE,
10101100110000010100011
```

Répresentation des fichiers permettant de visualiser l'arbre de codage de manière graphique:

Premier:



Dernier:



2.3. Programme 3: huffman.exe

2.3.1. main.c

Ce code décode un texte codé Huffman à partir du fichier géneré par le programme 2. Il inclut d'abord les fichiers d'en-tête nécessaires de la partie 2 et déclare une fonction.

La fonction principale commence par définir certaines variables et ouvrir un fichier pour la lecture. Il lit le fichier ligne par ligne et stocke certaines informations dans un vecteur de cases. Il rembobine ensuite le fichier et le lit à nouveau, décodant cette fois le texte codé par Huffman et imprimant le texte en clair résultant sur la console.

Le processus de décodage fonctionne en parcourant le texte codé Huffman et en stockant le code pour chaque caractère dans une chaîne. Lorsqu'un caractère dans le vecteur de boîtes est trouvé qui a le même code que la chaîne, le caractère est imprimé sur la console et la chaîne est réinitialisée à une chaîne vide. Ce processus se poursuit jusqu'à ce que la fin du texte codé a lieu. La partie de décodification a été un défi, avec des problèmes dans la partie final du code, mais à la fin on a réussi à le faire.

```c
int main(void) {
  char file[] = "archive_test.txt";
  printf("Opening file: %s \n",file);
  FILE *stream = fopen(file, "r");
  printf("Opened file: %s \n",file);
  char phrase_bin[200];
  char endoftable[1024] = "END OF TABLE";
  char caract;

  int m=0;
  char line[1024];
  while (fgets(line, 1024, stream)){
    m++;
  }
  m=m-2;
  rewind(stream);

  T_box vecteur_lettres[m];

  for(int j=1; j<=3; j++){
    int i = 0;
    int space = 0;
    char line[1024];

    while (fgets(line, 1024, stream)){

      char* tmp = strdup(line);

      if(j==1){
        if(i<m){
          vecteur_lettres[i].letre = getfield(tmp, j)[0];
        }
      }
      if(j==3){
        if(i<m){
          strcpy(vecteur_lettres[i].bin_str, getfield(tmp, j));
          printf("%s\n", vecteur_lettres[i].bin_str);
        }
      }
      //printf("Element %d would be %s\n", i, getfield(tmp, j));
      if(space == 1){
        strcpy(phrase_bin, getfield(tmp, j));
        printf("Frase em binário: %s \n", phrase_bin);
        space = 0;
        printf("space convertido para 0 \n");
      }
      if(j==1){
        if((strcmp(getfield(tmp, j), endoftable)==0)){
        // enregistrer le code binaire
        space = 1;
        printf("space convertido para 1 \n");
        }
      }
      free(tmp);
      i++;
    }
    rewind(stream);
  }

  int longueur_vec_l = sizeof(vecteur_lettres)/sizeof(vecteur_lettres[0]);
  //printf("%d\n", longueur_vec_l);

  // DECODIFICATION
  //char lettre_bin[phrase_bin];
  char *lettre_bin = calloc(strlen(phrase_bin), sizeof(char));
  char str_l_bin[15];
  //char str_l_bin[10];
  char phrase_decod;

  //printf("%s\n", lettre_bin);
  int counter = 0;
  for(int i=0; i<strlen(phrase_bin); i++) { // parcourir la string
    caract = phrase_bin[i];
    //printf("Caracter sendo adicionado: %c \n", caract);
    // printf("%s\n", lettre_bin);
    lettre_bin[counter] = caract;
    counter++;
    //strcat(lettre_bin, &caract); //concaténer les chaînes pour former les lettres
     //printf("%s\n", lettre_bin);
    //printf("String lettre_bin concatenada: %s\n",  lettre_bin);
    //break;
```

```
105 ▾   for(int j=0; j<longueur_vec_l;j++) { //parcourir le vecteur avec les structs des lettres
106
107 ▾     if (strcmp(lettre_bin, vecteur_lettres[j].bin_str) == 0) {
108           printf("%c", vecteur_lettres[j].letre);
109           memset(lettre_bin, '\0', strlen(phrase_bin)*sizeof(char));
110           counter = 0;
111           break;
112       }
113
114 ▾     /*
115       for(int k=0; k<(sizeof(vecteur_lettres[j])/sizeof(vecteur_lettres[j].bin));k++) {
116         if(vecteur_lettres[j].bin[k]!=-1){
117           snprintf(str_l_bin, 10, "%d", vecteur_lettres[j].bin[k]); //convertir le code binaire en chaîne
118         }else{break;}
119       }
120
121       if(strcmp(str_l_bin, lettre_bin) == 0){ //s'il y a déjà une lettre qui correspond au code assemblé
122         phrase_decod = *strcat(&phrase_decod, &vecteur_lettres[j].letre); //assembler la phrase décodée avec la lettre
123         lettre_ok = 1; // lettre trouvé
124         lettre_bin = ""; //
125         break; //
126       }
127
128       */
129     }
130
131
132   }
133
134   printf("\n");
135   return 0;
136 }
```

2.4. Output

```
> ./main
Opening file: archive_test.txt
Opened file: archive_test.txt
Phrase codifié en binaire: 0101001110011001011110
Phrase decodifié: abbacadabra
```

3. Conclusion et Perspectives

3.1. Sur le programme:

Un problème commun auquel le groupe a été confronté lors de l'écriture du code était l'éxécution correcte de l'algorithme. Le codage de Huffman consiste à créer un arbre binaire basé sur la fréquence des caractères dans un message, puis à utiliser l'arbre pour compresser le message. Ce processus a été difficile à conceptualiser, en particulier pour parce qu'il faut des techniques de compression de données.

Un autre problème rencontré par le groupe était de déterminer la meilleure façon de stocker l'arbre de Huffman. L'arbre devant être parcouru pour décoder le message, il est important qu'il soit accessible rapidement et efficacement. Cependant, le stockage de l'arborescence en mémoire prendre espace. Le groupe a dû évaluer le compromis entre l'utilisation de la mémoire et la vitesse de décodage au moment de décider comment stocker l'arbre.

Le programme a eu problèmes pour génèrer aussi des fichiers qui peuvent être utilisés pour visualiser graphiquement l'arbre partiellement ordonné et l'arbre de codage à l'aide de l'outil graphviz.

En général, l'écriture du code était une tâche difficile pour le groupe. Cela nécessitait une compréhension approfondie de l'algorithme, une réflexion approfondie sur la façon de stockage et une solution pour gérer les visualisations graphiques. Malgré ces défis, le groupe a finalement été en mesure de produire un code qui comprimait efficacement les messages à l'aide de la technique de codage Huffman.

3.2. Sur l’organisation du programme et celle du groupe:

Le fil rouge a été réalisé au cours des vacances, avec quelques jours réservés pour étudier le sujet et approfondir la compréhension de la matière. Une fois cette recherche initiale terminée, le reste des vacances a été consacré à l'écriture et au test du code. Cette approche a permis une réalisation approfondie et complète de tout, résultant en un travail avec plus de qualité.

En tant que groupe, nous avons décidé de consacrer nos vacances à l'écriture du code qui applique le codage Huffman. Afin de nous assurer que nous sommes en condition de bien faire cette tâche de manière efficace et efficiente, nous avons élaboré un plan détaillé pour nous guider tout au long du processus.

Tout d'abord, nous avons commencé par rechercher le codage Huffman et améliorer nos connaissances sur le sujet. Cela signifie surtout lire l'algorithme, ses principes et ses applications. On a passé également du temps à examiner des exemples de codage Huffman en action et à nous familiariser avec la langage C sur cela.

Une fois qu’on avait une solide compréhension du codage Huffman, nous avons commencé à réfléchir à des idées pour notre code. Il y a eu une discussion sur les exigences spécifiques du fil rouge, ainsi que des caractéristiques ou fonctionnalités supplémentaires que nous aimerions inclure. Nous avons examiné également tous les défis ou obstacles potentiels auxquels nous pourrions être confrontés au cours du processus de développement et on a proposé des stratégies pour les surmonter.

Ensuite, on a reparti le travail entre les membres du groupe en fonction de nos forces et intérêts individuels. Cela a garanti que chaque membre est en mesure de contribuer au projet de manière significative et que la charge de travail est répartie équitablement. En gros, Vitória et Rebeca ont principalement travaillé sur les parties 1 et 2 du code, tandis que David et Bernardo ont travaillé principalement sur la partie 3 du code et sur Compte Rendu. Il est à noter qu'au final tout le monde a travaillé sur toutes les parties du travail puisqu'au final avec les problèmes de code, il a fallu l'effort de tout le groupe pour les résoudre.

Une fois qu’on avait une idée claire de ce sur quoi chaque membre travaillerai, on a commencé le processus de codage proprement dit. On a réservé des blocs de temps dédiés chaque jour pour travailler sur le projet, dans le but de progresser régulièrement vers un fil rouge complet. Ensuite, on a planifié des réunions régulières avec le groupe pour partager nos progrès et discuter les problèmes qu’on a trouvé.

Pour nous assurer que notre code est de haute qualité, on a reservé également du temps pour les tests et le débogage. Cela implique d'exécuter notre code à travers une série de tests pour identifier les erreurs ou les problèmes, puis de travailler pour les résoudre. Après cela, on a essayé de bien documenter notre code, afin qu'il soit clair et facile à comprendre. Enfin, une fois le code terminé et entièrement testé, on a mis sur moodle.

Le groupe a bien travaillé ensemble sur le fil rouge et tout le monde a participé activement à la réalisation de la tâche et équitablement à toutes les étapes du travail (les 4 membres de l’équipe). Le groupe a pu répartir efficacement le travail et utiliser ses forces individuelles afin de progresser dans la tâche. La communication au sein du groupe était ouverte et efficace, car chacun a pu apporter ses idées et collaborer vers un objectif commun → le développement des 3 programmes. Alors, les efforts du groupe ont permis de compléter la mission à un niveau satisfaisant.

4. Réferences

- "The C Programming Language" by Brian Kernighan and Dennis Ritchie
- "C: A Reference Manual" by Samuel P. Harbison and Guy L. Steele
- "Programming in C" by Stephen G. Kochan
- "C: An Advanced Introduction" by Narain Gehani
- "Mastering C" by Herbert Schildt
- "Huffman Coding" by David Salomon
- "A Beginner's Guide to Huffman Coding" by J.G. Sullivan
- "Huffman Coding: An Algorithm for Data Compression" by John Gilbert
- "Coding the Huffman Algorithm in C" by Michael Schidlowsky
- "Compression and Huffman Coding in C" by Mark Nelson
- TEA3
- AAP - Séance 4
- AAP - Séance 5