Big Data | TP 1

> Presenté par: Omar ALLOUCH

Partie 1

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

int hash_function(char *word, int table_size) {
  return 0;
}

int main(int argc, char *argv[]) {
  FILE *file = fopen("texte Shakespeare.txt", "r");

  if (file == NULL) {
    printf("Error: Could not open file\n");
    return 1;
  }

  char word[100];
  while (fgets(word, sizeof(word), file)) {
    hash_function(word, 1000);
  }

  fclose(file);
  return 0;
}
```

Dans cette première version du code,one lit le contenu du fichier *word.txt* contenant nos mots, et on applique le fonction `hash_function` qui va nous servir plus tard à calculer la valeur de hachage de nos mots.

Partie 2

En regardant le fichier *word2.txt*, et à l'aide du taux d'occupation donné, on peut déduire la taille de notre table de hachage en utilisant la formule :

$$taux = \frac{|E|}{M}$$

On va de plus trouver le nombre premier le plus proche à cette valeur calculée pour aider à réduire les collisions et à améliorer la distribution.

Pour accomplir ça, on utiilise les fonctions suivantes :

```
# include <math.h>

# define MAX_OCCUPANCY_RATE 0.30
# define INITIAL_SIZE 235885

int is_prime(int n) {
  if (n <= 1) {
    return 0;
  }
  for (int i = 2; i <= sqrt(n); i++) {
    if (n % i == 0) {
      return 0;
    }
  }
  return 1;
}

int next_prime(int n) {
  while (!is_prime(n)) {
    n++;
```

```
  }
  return n;
}

int calculate_table_size(int distinct_words) {
  int table_size = (int)(distinct_words / MAX_OCCUPANCY_RATE);
  return next_prime(table_size);
}

int main(int argc, char *argv[]) {
    ...
  int table_size = calculate_table_size(INITIAL_SIZE);
  printf("Table size: %d\n", table_size);
  ...
}
```

Partie 3

Voici une simple fonction de hachage avec de bonnes propriétés générales :

```
int hash_function(char *word, int table_size) {
  int hash = 4291;
  for (int i = 0; i < strlen(word); i++) {
    hash ^= (hash << 5) + (hash >> 2) + word[i];
  }
  return hash % table_size;
}
```

On prend une valeur initiale de `hash`, puis on parcourt la chaîne de caractères et on applique divers opérations sur les bits.

Partie 4

Vu que le but est de simplement compter le nombre de collisions produites suite à l'application de notre fonction de hachage, on a opté à utiliser une simple table contenant dans 0 et des 1 pour savoir si une valeur de hachage a déjà été utilisée.

```
int main(int argc, char *argv[]) {
  ...
  __uint8_t table[table_size];
  for (int i = 0; i < table_size; i++) {
    table[i] = 0;
  }

  int collisions = 0;
  char word[100];
  while (fgets(word, sizeof(word), file)) {
    int hash = hash_function(word, table_size);
    if (table[hash] != 0) {
      collisions++;
    } else {
      table[hash] = 1;
    }
  }
  ...
}
```

Les indicateurs de performances auquels qu'on va s'intéresser sont le nombre de collisions (le plus important), et le temps d'exécution du programme (en utilisant la commande `time`).

Partie 5

Voici une autre fonction de hachage :

```
int other_hash_function(char *word, int table_size) {
  int hash = 33;
  for (int i = 0; i < strlen(word); i++) {
    hash += word[i];
    hash *= (hash << 4);
    hash ^= (hash >> 10);
  }
```

```
    return hash % table_size;
}
```

Partie 6

	Fonction 1 (`hash_function`)	Fonction 2 (`other_hash_function`)
# Collisions (Shakespeare)	183	397
# Collissions (Corncob)	1033	2588
Temps d'exécution	~0.01s	~0.01s

On peut voir qu'il existe presque aucune différence dans le temps de traitement entre les deux fonctions de hachage. Par contre, la première fonction de hachage génère presque 2,5 fois moins de collisions que la seconde.

On peut dire que la première fonction a une meilleure distribution des valeurs produites dans l'espace de hachage.