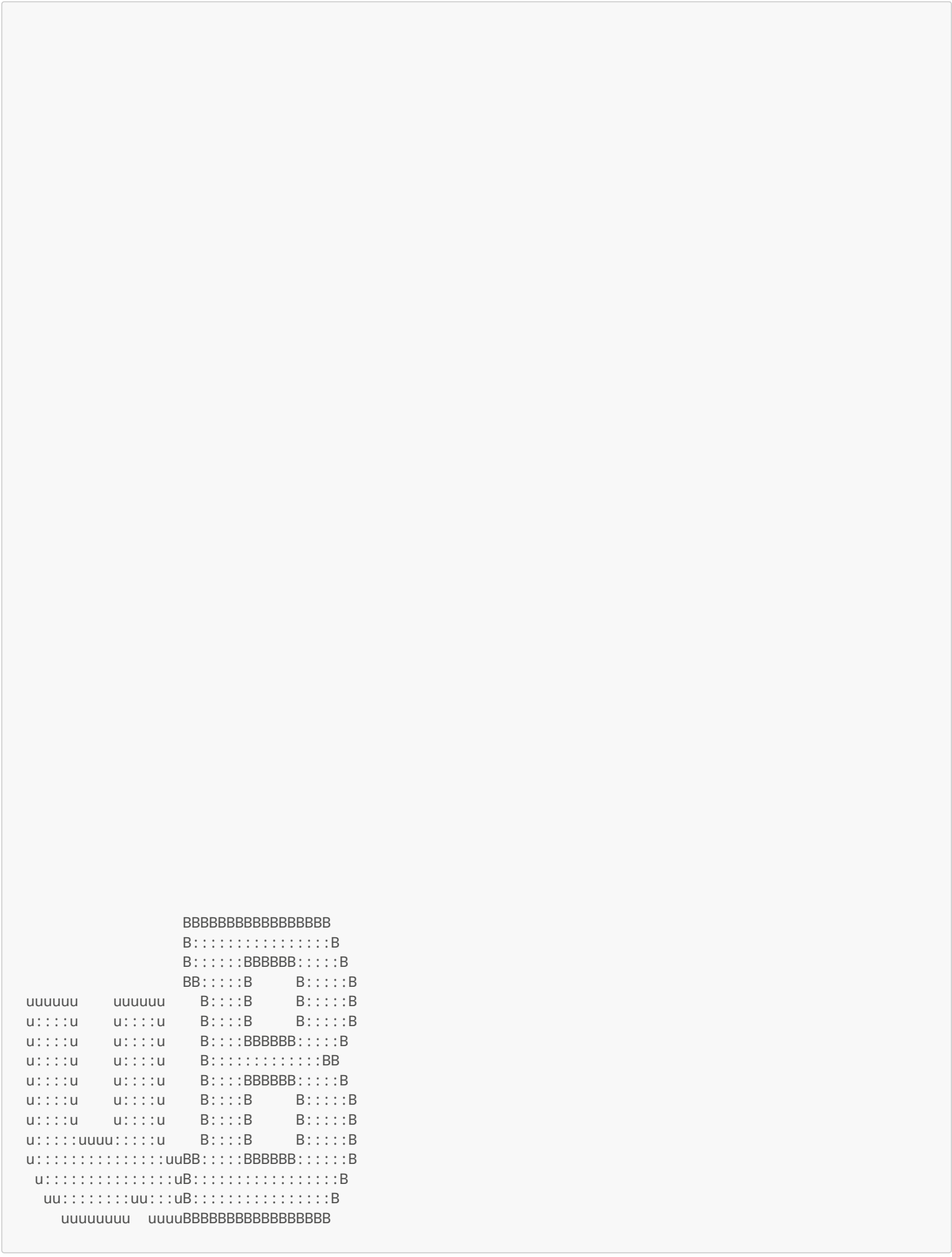
Guillaume BELDILMI

Projet Codage et Cryptographie



Introduction

L'objectif de ce projet était de un programme mettant en oeuvre des notions de codage et de cryptographie vues en cours. Nous devions reproduire les traitements réalisés par David Albert Huffman sur une lettre codée en binaire lui ayant été envoyée par Richard Hamming afin de la déchiffrer pour enfin la envoyer à son destinataire final après l'avoir chiffrée à nouveau puis compressée.

Le choix du langage de programmation était libre, nous avons choisi de réaliser ce projet en Rust. Ce langage, n'étant pas enseigné en cours, nous a permis d'allier cet exercice à un apprentissage personnel.

Etape 1 : Détecter et corriger les erreurs

La lettre codée en binaire a été envoyée par Richard Hamming, mais des erreurs de transmission se sont produites. Afin de les corriger, nous devions connaître le code de Hamming utilisé pour coder la lettre.

Nous savons que le codage a rendu la lettre plus longue d'un facteur 1,75. Ainsi, pour `x` bits de données, il y a `y = 0.75 * x` bits de contrôle. Un bit ne pouvant être divisé, les options possibles pour un nombre minimal de bits de données sont `x = 4` et ses multiples.

Pour notre première hypothèse (segmentation la plus petite possible), nous avons donc `x = 4` et `y = 3` bits de contrôle, ce qui nous donne des mots de Hamming de 7 bits. Pour confirmer cette hypothèse, nous avons vérifié que la longueur de la lettre codée en binaire était bien un multiple de 7, or `47712 = 6816 * 7`.

Les mots de Hamming sont des mots de `n` bits désignant un alphabet limité de moins de `2^n` symboles. Or, dans notre cas, nous savons que la lettre transmise contient deux erreurs (une au début et une à la fin). Pour vérifier que notre segmentation est correcte, nous avons donc divisé la lettre en mots de 7 bits et compté le nombre d'itérations de chaque occurrence avec la commande suivante :

```
cat doc/lettre.txt | grep -o -E '.{7}' | sort | uniq -c | sort -n
```

Ce qui nous a donné le résultat suivant :

```
      1 0011101
      1 1100111
     57 1111111
     75 1101010
     79 1011000
    105 1110100
    142 1010011
    155 0101100
    175 1001101
    200 0001011
    212 0100111
    213 1100001
    303 1000110
    329 0011110
    688 0000000
    982 0010101
   1359 0111001
   1740 0110010
```

Nous avons donc bien deux occurrences de 7 bits qui ne sont pas répétées (`0011101` et `1100111`), ce qui confirme notre hypothèse. Nous avons donc décidé de continuer avec cette segmentation.

Afin de pouvoir correctement corriger ces erreurs, nous devons connaitre comment les bits de contrôle sont calculés. Pour cela, nous avons utilisé un cas récurrent vu en cours : le code de Hamming (7,4,3). Dans ce cas, les bits de contrôle sont calculés de la manière suivante, pour sept bits de données `c1 c2 c3 c4 c5 c6 c7` :

- `c1`, `c2`, `c3` et `c4` sont les bits de données ;
- `c5` est le bit de parité pour les bits `c1`, `c2` et `c3` ;
- `c6` est le bit de parité pour les bits `c1`, `c2` et `c4` ;
- `c7` est le bit de parité pour les bits `c2`, `c3` et `c4`.

Nous avons donc implémenté une fonction en Rust permettant de détecter ces erreurs. Cette fonction prend en paramètre les sept bits de données et retourne un entier correspondant à l'erreur détectée. Si aucune erreur n'est détectée, la fonction retourne 0. Voici le code de cette fonction :

```rust
fn hamming7(c1: u8, c2: u8, c3: u8, c4: u8, c5: u8, c6: u8, c7: u8) -> usize {
  let p1 = c1 ^ c2 ^ c3; // first control bit
  let p2 = c1 ^ c2 ^ c4; // second control bit
  let p3 = c2 ^ c3 ^ c4; // third control bit
  match (c5 != p1, c6 != p2, c7 != p3) { // check control bits
    (true,  true,  false) => { 1 }, // error in c1
    (true,  true,  true)  => { 2 }, //    "     c2
    (true,  false, true)  => { 3 }, //    "     c3
    (false, true,  true)  => { 4 }, //    "     c4
    _                     => { 0 }  // no error
  }
}
```

Si une erreur est détectée, il nous suffira juste d'inverser le bit erroné pour la corriger.

```rust
fn correct_errors(input: Vec<u8>) -> Vec<u8> {
  let mut corrected = Vec::new();
  let mut i:usize = 0;
  while i + 7 < input.len() { // while there are 7 bits to read
    // get the index of the error
    let c = hamming7(input[i], input[i+1], input[i+2], input[i+3], input[i+4], input[i+5], input[i+6]);
    for j in 0..7 {
      corrected.push(
        if j + 1 != c { // if the bit is not the error
          input[i+j]
        } else { // if the bit is the error, correct it
          input[i+j] ^ 0x01
        });
    }
    i += 7; // move to the next 7 bits
  }
  corrected
}
```

Le résultat de cette étape apparaîtra dans le fichier `--1-corrected.txt`.

Etape 2 : Supprimer les bits de contrôle

Une fois les erreurs corrigées, nous devons supprimer les bits de contrôle pour retrouver uniquement les bits de données. Pour cela, nous avons implémenté une fonction en Rust qui prend en paramètre un vecteur de bits et qui copie ces bits dans un nouveau vecteur en supprimant les bits de contrôle. Voici le code de cette fonction :

```rust
fn reduce(input: Vec<u8>) -> Vec<u8> {
  let mut reduced = Vec::new();
  let mut i:usize = 0;
  while i + 7 <= input.len() {  // while there are 7 bits to read
    for j in 0..4 {             // read the first 4 bits
      reduced.push(input[i+j]); // and add them to the reduced vector
    }
    i += 7; // move to the next 7 bits
  }
  reduced
}
```

Le résultat de cette étape apparaîtra dans le fichier `--2-reduced.txt`.

Etape 3 : Traduire les bits en caractères alphanumériques

Une fois les bits de données récupérés, nous devons les traduire en caractères alphanumériques. Voici le code de la fonction permettant de réaliser cette traduction :

```rust
fn convert_to_ascii(input: Vec<u8>) -> String {
  let s = String::from_iter(group_bytes(&input).iter().map(|v| { *v as char }));
```

```
  s
}
```

Chaque élément du vecteur `input` représentant un bit, nous devons regrouper ces bits par 8 pour former des octets. Pour cela, nous avons implémenté la fonction `group_bytes` :

```
fn group_bytes(bytes: &Vec<u8>) -> Vec<u8> {
  let mut grouped = Vec::new();
  let mut i:usize = 0;
  let mut b:u8 = 0;
  for byte in bytes {
    b = b << 1 | byte; // add the bit to the byte
    i += 1;
    if i == 8 {        // if the byte is complete
      grouped.push(b); // add it to the grouped vector
      b = 0;
      i = 0;
    }
  }
  grouped
}
```

Une fois les octets formés, nous pouvons convertir chaque octet (`u8`) en caractère alphanumérique simplement en castant chaque élément du vecteur en `char` et en les regroupant dans une chaîne de caractères.

Le résultat de cette étape apparaîtra dans le fichier `--3-ascii.txt`.

Etape 4 : Déchiffrer la lettre

Une fois la lettre traduite en caractères alphanumériques, nous obtenons le message suivant :

```
Ac 02 fhff 1952, p Knyfnn Fbsz, N fsb ks qgmba, Qutp Tsoa Iskpbt, [...]
```

Pour déchiffrer cette lettre, nous devons connaître le chiffrement ainsi la clé de chiffrement utilisés. Le sujet nous indique que la lettre a été chiffrée avec un chiffrement polyalphabétique du XVIe siècle. Une recherche sur internet nous a rapidement orienté vers le chiffrement de Vigenère. Or, ce chiffrement s'appliquant sur des lettres, nous supposerons que chaque caractère différent d'une lettre sera ignoré par le chiffrement (ce qui laisse du sens aux chiffres, aux espaces et à la ponctuation dans le message chiffré).

Afin de pouvoir déchiffrer la lettre, nous avons implémenté une fonction en Rust permettant de déchiffrer, à partir d'une clé définie, un message chiffré avec le chiffrement de Vigenère. Voici le code de cette fonction :

```
fn decrypt(input: String) -> String {
  let key = "python";
  let diff: Vec<u8> = key.chars().map(|c| { c as u8 }).collect(); // convert the key to a vector of
bytes
  let mut decrypted = String::new();
  let mut i: usize = 0;
  for c in input.chars() { // for each character in the input
    // if the character is a letter (adapt the case if necessary)
    let b = if c >= 'a' && c <= 'z' {
      'a' as u8
    } else if c >= 'A' && c <= 'Z' {
      'A' as u8
    } else {
      0
    };
    if b != 0 { // if the character is a letter (decrypt it)
      let l = c as u8 - b;
      let k = diff[i] - 'a' as u8;
      let d = (l + 26 - k) % 26 + b;
      decrypted.push(d as char);
      i = (i + 1) % key.len();
    } else { // if the character is not a letter (keep it as is)
      decrypted.push(c);
```

```
      }
    }
    decrypted
  }
```

Réaliser une attaque par force brute serait fastidieuse, nous allons donc essayer d'utiliser une attaque par dictionnaire. De plus, afin de réduire le domaine des clés à tester, nous utiliserons des indices laissés dans le sujet, ainsi que la sémantique supposée du message.

Nous ne connaissons pas la clé de chiffrement utilisée hormis le fait qu'elle soit liée au domaine des serpents. Ne connaissant pas la langue dans laquelle la lettre a été écrite, nous avons décidé de considérer les langues les plus probables : l'anglais (langue maternelle de Claude Shannon, Richard Hamming et David Albert Huffman qui sont tous trois américains) et le français (langue dans lequel cette unité d'enseignement est dispensée).

En recoupant ces informations, nous pouvons supposer qu'une clé de chiffrement évidente serait `serpent` ou `snake` en anglais. Nous avons donc décidé de tester ces deux clés pour déchiffrer la lettre.

- Avec la clé `serpent`, nous obtenons le message suivant : `Iy 02 osbs 1952, w Sjhqja Mjoi, Y bfi so zrioh, Yqca Pfvi Ebvloa, [...]`
- Avec la clé `snake`, nous obtenons le message suivant : `Ip 02 fxbn 1952, c Kdunan Vxam, N voj xs gcuoa, Gqbc Tiki Vsaljg, [...]`

Ces deux résultats n'ayant pas de sens, ces clées sont donc incorrectes. Nous pourrions continuer à tester des clés de chiffrement correspondant à différentes espèces de serpents en anglais et en français. Cependant, il sera encore pertinant de réduire encore le domaine des clés à tester grâce à la sémantique supposée du message.

Le texte à déchiffrer, étant une lettre, nous pouvons supposer qu'au début d'une lettre, il puisse apparaître : une date, un nom, une adresse, une formule de politesse, etc.

Or, le message commence par `Ac 02 fhff 1952, [...]`, "02" et "1952" semblant composer une date, nous pouvons supposer que le mot `fhff` désigne un mois. Etant composé de quatre lettres, nous pouvons supposer que ce mois peut être `mars`, `juin` ou `aout` en français ou `june` ou `july` en anglais.

Pour nous aider, nous avons utilisé la table de Vigenère suivante (la première ligne correspond à la lettre du message et la première colonne à la lettre de la clé, la lettre chiffrée est donc l'intersection de la ligne et de la colonne correspondant à la lettre du message et de la clé) :

Clé \ Message	**A**	**B**	**C**	**D**	**E**	**F**	**G**	**H**	**I**	**J**	**K**	**L**	**M**	**N**	**O**	**P**	**Q**	**R**	**S**	**T**	**U**	**V**	**W**	**X**	**Y**	**Z**
A	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
B	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A
C	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B
D	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C
E	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D
F	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E
G	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F
H	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G
I	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H
J	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I
K	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J
L	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K
M	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L
N	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M
O	O	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N
P	P	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
Q	Q	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
R	R	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q

Clé \ Message	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z
S	S	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R
T	T	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S
U	U	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T
V	V	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U
W	W	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
X	X	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W
Y	Y	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X
Z	Z	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y

En testant les clés `mars`, `juin` et `aout` en français et `june` et `july` en anglais, nous obtenons les résultats suivants concernant les clés possibles :

- Si `fhff` correspond à `mars`, nous obtenons le morceau de clé : `..thon..`
- Si `fhff` correspond à `juin`, nous obtenons le morceau de clé : `..wnns..`
- Si `fhff` correspond à `aout`, nous obtenons le morceau de clé : `..ftlm..`
- Si `fhff` correspond à `june`, nous obtenons le morceau de clé : `..wnsb..`
- Si `fhff` correspond à `july`, nous obtenons le morceau de clé : `..wnuh..`

Nous pouvons donc supposer que le mot `fhff` correspond à `mars` et que la clé de chiffrement et donc le contenu du message sont en français car le morceau de clé obtenu est le plus cohérent parmi les résultats obtenus.

Nous avons donc identifié la langue du message et un morceau de la clé de chiffrement. Nous pouvons donc continuer notre analyse avec le premier mot du message chiffré `Ac` qui pourrait être un article précédant une date. Nous pouvons donc supposer que le mot `Ac` pourrait correspondre à l'article `Le`, or cela nous indiquerait que la clé de chiffrement commence par les caractères `py...`.

En mettant en correspondance les lettres du message chiffré avec les lettres du message déchiffré, nous obtenons le morceau de clé suivant : `python`, qui est bien une espèce de serpent. Nous avons donc essayé de déchiffrer le message avec cette clé et nous obtenons le début de message suivant :

```
Le 02 mars 1952, a Murray Hill, A qui de droit, Cher Alan Turing, [...]
```

Nous avons donc réussi à déchiffrer la lettre. Le résultat de cette étape apparaîtra dans le fichier `--4-decrypted.txt`.

Etape 5 : Chiffrer la lettre

Afin de chiffrer la lettre, nous devons utiliser une variante du chiffrement de Vigenère. En effet, le chiffrement de Vigenère est un chiffrement polyalphabétique qui utilise une clé pour chiffrer un message. Cependant, ce chiffrement est vulnérable à une attaque par force brute. Pour pallier à cette faiblesse, nous allons utiliser un chiffrement de Vigenère avec une clé aléatoire de longueur égale à celle du message à chiffrer.

Pour cela, nous avons implémenté une fonction en Rust chiffrant ce message avec une clé aléatoire créée à partir de la fonction `random` de la bibliothèque standard de Rust. Voici le code de cette fonction, elle prend en paramètre le message à chiffrer et retourne le message chiffré ainsi que la clé de chiffrement utilisée :

```rust
fn encrypt(input: String) -> (Vec<u8>, Vec<u8>) {
  let bin = convert_to_bin(input); // convert the input to a vector of bits
  let mut encrypted = Vec::new();
  let mut token = Vec::new();
  for b in bin {                   // for each bit in the input
    let k = random::<u8>() & 0x01; // generate a random bit
    encrypted.push(b ^ k);         // add the encrypted bit to the encrypted vector
    token.push(k);                 // add the token bit to the token vector
  }
  (encrypted, token) // return the encrypted vector and the token vector
}
```

Notons que dans notre cas, nous chiffrons le message en binaire, permettant de chiffrer les caractères alphanumériques ainsi que les caractères spéciaux, rendant le message chiffré plus difficile à déchiffrer.

La fonction `convert_to_bin` permet de convertir le message (une chaine de caractères) en une suite de bits. Cette fonction est définie comme suit :

```
fn convert_to_bin(input: String) -> Vec<u8> {
  let mut bin = Vec::new();
  for c in input.as_bytes() {       // for each byte in the input
    for i in 0..8 {                 // for each bit in the byte
      bin.push((c >> (7 - i)) & 1); // add the bit to the bin vector
    }
  }
  bin
}
```

Le résultat de cette étape apparaîtra dans le fichier `--5-encrypted.txt`, sa clé de chiffrement apparaîtra dans le fichier `--5-token.txt`.

Etape 6 : Compresser la lettre sa clé de chiffrement

Afin de compresser la lettre chiffrée et sa clé de chiffrement, nous devons utiliser un algorithme de compression. Pour cela, nous avons décidé d'utiliser l'algorithme de Huffman. Cet algorithme permet d'obtenir le codage binaire optimal d'un message en remplaçant les occurences les plus fréquents par des codes binaires plus courts. Dans notre cas, nous allons compresser un message binaire, nous allons donc segmenter le message en octets et obtenir le codage optimal de ces derniers.

L'algorithme de Huffman est composé de plusieurs étapes :

- Tout d'abord, il faut compter le nombre d'occurrences de chaque octet dans le message à compresser.
- Ensuite, il faut créer un arbre binaire de Huffman à partir de ces occurrences.
- Enfin, il faut parcourir l'arbre pour obtenir le codage binaire de chaque octet.
- Et pour finir, il faut remplacer chaque octet par son codage binaire optimal correspondant.

Cela nous donne le code suivant :

```
fn compress(input: Vec<u8>) -> (String, String) {
  let data = group_bytes(&input); // group the bytes by 8
  let stats = stats(&data);       // get statistics on the values of the data (value, frequency)
  let huffman = huffman(&stats);  // get the Huffman encoding of the values (value, encoding)
  let mut compressed = String::new();
  for byte in data {                    // for each byte in the data
    for (value, encoding) in &huffman { // for each value and its encoding
      if byte == *value {               // if the byte is the value
        compressed.push_str(encoding);  // add the encoding to the compressed string
        break;
      }
    }
  }
  // return the compressed string and the encoding string
  (compressed, encoding_to_string(&stats, &huffman))
}
```

Nous avons déjà vu la fonctions `group_bytes()` précédemment, nous détaillerons le reste des fonctions utilisées dans cette étape dans les paragraphes suivants.

Afin d'obtenir le nombre d'occurrences de chaque valeur d'octet, nous avons implémenté une fonction en Rust permettant de compter ces occurrences pour chaque valeur que peut prendre l'octet. Voici le code de la fonction `stats()` :

```
fn stats(data: &Vec<u8>) -> Vec<(u8, u128)> { // (value, number of occurrences)
  let mut stats = Vec::new();
  for i in 0..256 {           // for each possible value
    stats.push((i as u8, 0)); // initialize the number of occurrences to 0
  }
  for i in data {              // for each value in the data
    stats[*i as usize].1 += 1; // increment the number of occurrences for this value
  }
  stats.sort_by(|a, b| b.1.cmp(&a.1)); // sort the values by decreasing number of occurrences
  stats
}
```

La fonction `stats()` retourne un vecteur de tuples contenant chacun la valeur de l'octet et le nombre d'occurrences de cette valeur dans le message à compresser. Ce vecteur est trié par ordre décroissant du nombre d'occurrences puis transmis à la fonction `huffman()` dont voici le code :

```
fn huffman(stats: &Vec<(u8, u128)>) -> Vec<(u8, String)> {
  let mut nodes = Vec::new();
  for (value, frequency) in stats { // for each value and its frequency, create a node
    let value = *value;             //  (availability, value, frequency, left child index, right child
index)
    let frequency = *frequency;
    nodes.push((true, value, frequency, 0, 0));
  }
  loop {
    let (min1, min2) = get_mins(&nodes); // get the indexes of the two nodes with the smallest
frequency
    if min1 == -1 || min2 == -1 { // if there is only one node left, stop, it is the root
      break;
    } else {
      let min1 = min1 as usize;
      let min2 = min2 as usize;
      let sum_values = nodes[min1].2 + nodes[min2].2; // create a new node with the sum of the
frequencies of the two nodes
      nodes.push((true, 0, sum_values, min1, min2));  // and add it to the nodes
      nodes[min1].0 = false;                          // mark the two nodes as unavailable
      nodes[min2].0 = false;
    }
  }
  // convert to (value, encoding)
  let root = get_mins(&nodes).0 as usize; // the root is the last node added
  let r = encode_node(&nodes, root, String::new()); // encode the nodes of the tree
  r
}
```

Notons que dans le cas présent, nous avons choisi de ne pas utiliser de structure de données pour représenter les noeuds de l'arbre de Huffman, mais un vecteur de tuples. Chaque tuple représente un noeud de l'arbre et est composé de cinq éléments : un booléen indiquant si le noeud est disponible (si ce dernier n'a pas de parent), la valeur de l'octet, le nombre d'occurrences de cette valeur, l'index du fils gauche et l'index du fils droit.

Dans le cas d'une feuille de l'arbre, les indices des fils gauche et droit seront égaux à 0. Dans le cas d'un noeud interne, ces indices correspondront aux indices des noeuds fils dans le vecteur `nodes`, la valeur de l'octet sera égale à 0 et le nombre d'occurrences sera égal à la somme des occurrences des noeuds fils.

La première boucle `for` permet de créer les noeuds feuilles de l'arbre de Huffman à partir des statistiques obtenues précédemment. La seconde boucle `loop` permet de lier les noeuds entre eux pour former l'arbre de Huffman. La fonction `get_mins()` permet de récupérer les indices des deux noeuds ayant les plus petites fréquences. Cette dernière fait appel à deux reprises à la fonction `get_min()` qui retourne l'indice du noeud ayant la plus petite fréquence, qui n'est pas encore lié à un autre noeud tout en pouvant spécifier un noeud à ignorer. Voici le code de ces deux fonctions :

```
fn get_min(nodes: &Vec<(bool, u8, u128, usize, usize)>, ignore: (bool, usize)) -> isize {
  let mut min = -1;  // -1 means no node found
  for j in 0..nodes.len() { // for each node
    if nodes[j].0 {         // if the node is available
      if (ignore.0 && j != ignore.1) || (!ignore.0) {        // if the node is not ignored
        if min == -1 || nodes[j].2 < nodes[min as usize].2 { // if the node has a smaller frequency
than the current minimum
          min = j as isize;                                 // update the minimum
        }
      }
    }
  }
  min
}

fn get_mins(nodes: &Vec<(bool, u8, u128, usize, usize)>) -> (isize, isize) {
  let mut mins = (-1, -1);
```

```
  mins.0 = get_min(&nodes, (false, 0));
  mins.1 = get_min(&nodes, (true, mins.0 as usize));
  mins
}
```

Si il ne reste qu'un seul noeud disponible, c'est que l'arbre est complet et que la racine est le dernier noeud ajouté. La fonction `huffman()` sort de sa seconde boucle et appelle la fonction `encode_node()` qui permet de parcourir l'arbre de Huffman pour obtenir le codage binaire de chaque valeur d'octet. Voici le code de cette fonction :

```
fn encode_node(nodes: &Vec<(bool, u8, u128, usize, usize)>, i: usize, prefix: String) -> Vec<(u8,
String)> {
  let mut encoding = Vec::new();
  if nodes[i].3 == 0 && nodes[i].4 == 0 { // Leaf node case (no child)
    encoding.push((nodes[i].1, prefix));
  } else {                              // Internal node case (two children)
    let mut left = prefix.clone();
    let mut right = prefix.clone();
    left.push('0');
    right.push('1');
    encoding.append(&mut encode_node(nodes, nodes[i].3, left)); // encode the left child
    encoding.append(&mut encode_node(nodes, nodes[i].4, right)); // encode the right child
  }
  encoding
}
```

La fonction `encode_node()` retourne un vecteur de tuples contenant chacun la valeur de l'octet et son codage binaire optimal trouvé. Une fois le résultat retourné, la fonction `compress()` remplacera chaque octet par son codage binaire optimal. Et pour finir, la fonction `encoding_to_string()` permet de convertir le vecteur de tuples en une chaîne de caractères listant, pour chaque valeur d'octet, sa fréquence et son codage binaire choisi afin d'avoir ainsi un suivi du résultat de la compression.

Le résultat de la compression du message apparaîtra dans le fichier `--6-compressed-message.txt`, le résultat de la compression de la clé de chiffrement apparaîtra dans le fichier `--6-compressed-token.txt` et leur encodage de Huffman apparaîtront dans les fichiers `--6-compressed-message-encoding.txt` et `--6-compressed-token-encoding.txt`.

Afin de vérifier l'efficaicité de notre compression, nous avons utilisé les commandes suivantes :

```
cat ./--5-encrypted.txt | wc -c
cat ./--6-compressed-message.txt | wc -c
```

Contrairement à ce qui était attendu, la taille du fichier n'a pas été significativement réduite. Cela est dû au fait que le message chiffré est proche d'un message aléatoire dû à la clé aléatoire utilisée pour le chiffrer en binaire. En effet, un message aléatoire ne peut pas être compressé efficacement par l'algorithme de Huffman car les valeurs d'octets sont presque toutes équiprobables.

Afin de vérifier que notre compression fonctionne correctement, nous avons réalisé un programme de test en Rust reproduisant toutes les étapes précédentes mais en omettant l'étape de chiffrement. Nous avons donc modifié la fonction `encrypt()` pour qu'elle utilise une clé de chiffrement composée uniquement de zéros, ainsi le message chiffré sera identique au message en clair avec les répétitions que cela implique. Nous avons ensuite compressé le message désormais non-chiffré.

Voici le code de la fonction `encrypt()` modifiée :

```
fn encrypt(input: String) -> (Vec<u8>, Vec<u8>) {
  let bin = convert_to_bin(input); // convert the input to a vector of bits
  let mut encrypted = Vec::new();
  let mut token = Vec::new();
  for b in bin {                   // for each bit in the input
    let k = 0;                     // the key is now fixed to 0
    encrypted.push(b ^ k);         // add the encrypted bit to the encrypted vector
    token.push(k);                 // add the token bit to the token vector
  }
  (encrypted, token) // return the encrypted vector and the token vector
}
```

Enfin, nous avons comparé la taille de notre message non-chiffré avec la taille du message compressé avec les même commandes utilisées précédemment. Les résultats obtenus ont été respectivement 27280 pour le message non-chiffré et 14622 pour le message compressé. Cela nous donne une réduction de près de la moitié de la taille du message initial, ce qui est un résultat très satisfaisant.

Conclusion

Nous avons pu réaliser toutes les étapes demandées dans le sujet, à savoir : la détection et la correction d'erreurs, la traduction des bits en caractères alphanumériques, le déchiffrement de la lettre, le chiffrement de la lettre, la compression de la lettre et de sa clé de chiffrement.

Ce projet nous a permis de mettre en pratique les notions de codage de l'information vues en cours ainsi que d'exercer nos compétences de programmation en Rust.