Baré Alexandre s172388
Wallon Robin s171708

Project 2 : Source Coding, Data Compression and Channel Coding

ELEN0060 - Information and Coding Theory

Université de Liege
Faculté des Sciences Appliquées
Année 2020-2021

1 Source coding and reversible (lossless) data compression

1.1 Question 1

To implement the binary Huffman code, we rely on a class Tree with 3 properties : a probability value and 2 references : one to the left child and the other to the right child. For each given probability, we create a new leaf node. Then, we loop over the following process until we obtain a full binary Huffman tree :

— Find the 2 nodes having the smallest probability values and no parent yet.

— Sum their value up and instantiate a new internal node with the result. The 2 nodes are assigned respectively to this new node's left and right child.

Afterwards, we have to traverse the tree from the root to each leaf adding to the codes in construction the value '0' for each left child and the value '1' for each right child. We do this via a recursive function that progressively concatenates the codes with the additional 0's and 1's.

We could easily extend this function to generate a Huffman code of arbitrary alphabet size by extending the class Tree we have implemented with a list of child references, instead of a reference to the left and right child. Then, our code would associate different values from the set $\{0, ..., alphabet\_size - 1\}$ to each direct child of the same parent. Traversing identically the tree as before would yield the final list of codes.

On exercise 7, we obtain the following output : ['101' '100' '000' '001' '11' '01'] where we can note that the smallest codes in terms of length are matching the highest probability values of the input vector [0.05,0.10,0.15,0.15,0.2,0.35]

1.2 Question 2

With our implementation of the on-line Lempel-Ziv algorithm, we obtain the following output on the example of the course (input = "1011010100010") : "100011101100001000010" with as dictionary : {'1' : 1, '0' : 2, '11' : 3, '01' : 4, '010' : 5, '00' : 6, '10' : 7} where the addresses are in decimal form.

1.3 Question 3

To make use of the basic Lempel-Ziv algorithm, it is necessary to know in advance the number of unique encoded entries $c(n)$ that we will store in the dictionary of codes. The integer addresses $\in [0, ..., c(n)]$ will be converted to binary and will have a fixed and common length for the whole encoding. We will thus have different codes depending on the number of entries to encode which itself depends on the input sequence and input length. These must therefore be read in advance. The basic Lempel-Ziv algorithm leads in practice to longer encoded texts because of the fixed-and-common-address-length constraint.

On the contrary, the on-line Lempel-Ziv algorithm deduces the length of the addresses based on the current number of encoded entries in the dictionary. The different addresses can therefore differ in length which enables on-line encoding. Indeed, it is not necessary to read the input sequence in advance anymore to encode (or decode). This approach is more robust as no assumption needs to be made about the source and it is also more flexible as it enables on-line encoding with a local dictionary.

In general, both methods are universal encoders but require a very long input sequence to benefit from the asymptotic convergence to the source entropy and are in general not competitive to tailored algorithms for specific source in terms of compression rate.

1.4 Question 4

See the code. Having implemented the algorithm so that the offset and the length are rewritten in fixed-length binary representations, we obtain for the input "abracadabrad" the output "000000a000000b000000r011001c010001d111100d", or equivalently with the offset and length in decimal : "(0,0,a)(0,0,b)(0,0,r)(3,1,c)(2,1,d)(7,4,d)".

1.5 Question 5

The total binary genome length is $n_1 = 7668456$. The total encoded genome length is $n_2 = 1885514$. The compression rate is thus $\frac{n_1}{n_2} = 4.06704$.

1.6 Question 6

The expected average length of the Huffman code is

$$\bar{n}_2 = \sum_i length(coded\_codon_i) * marginal\_probability(coded\_codon_i) = 5.901$$

.

(a)

The number of codons is $N = \frac{n_1}{\bar{n}_1} = \frac{958557}{3} = 319519$. The empirical average length of the Huffman code is $\frac{n_2}{N} = \frac{1885514}{319519} = 5.901$ for the full genome encoding. In the case of the full genome encoding, it is thus converging to the expected average length.

(b)

The (average) codon length is $\bar{n}_1 = 3$. The alphabet size of the encoded genome is $q_2 = 2$.

The theoretical bounds for the average length of any Huffman code are $\frac{H(S)}{\log_2(q_2)} \leq \bar{n}_2 < \frac{H(S)}{\log_2(q_2)} + 1$. In our case, $q_2 = 2$ and $H(S) = -\sum_i^N marginal\_probability(coded\_codon_i) * \log_2(marginal\_probability(coded\_codon_i)) = 5.867$. The average length is thus $\in [5.867; 6.867[$. Our expected average length falls perfectly in this interval and is moreover quite close to the lower bound, i.e. quite close to the absolutely optimal code average length.

1.7 Question 7

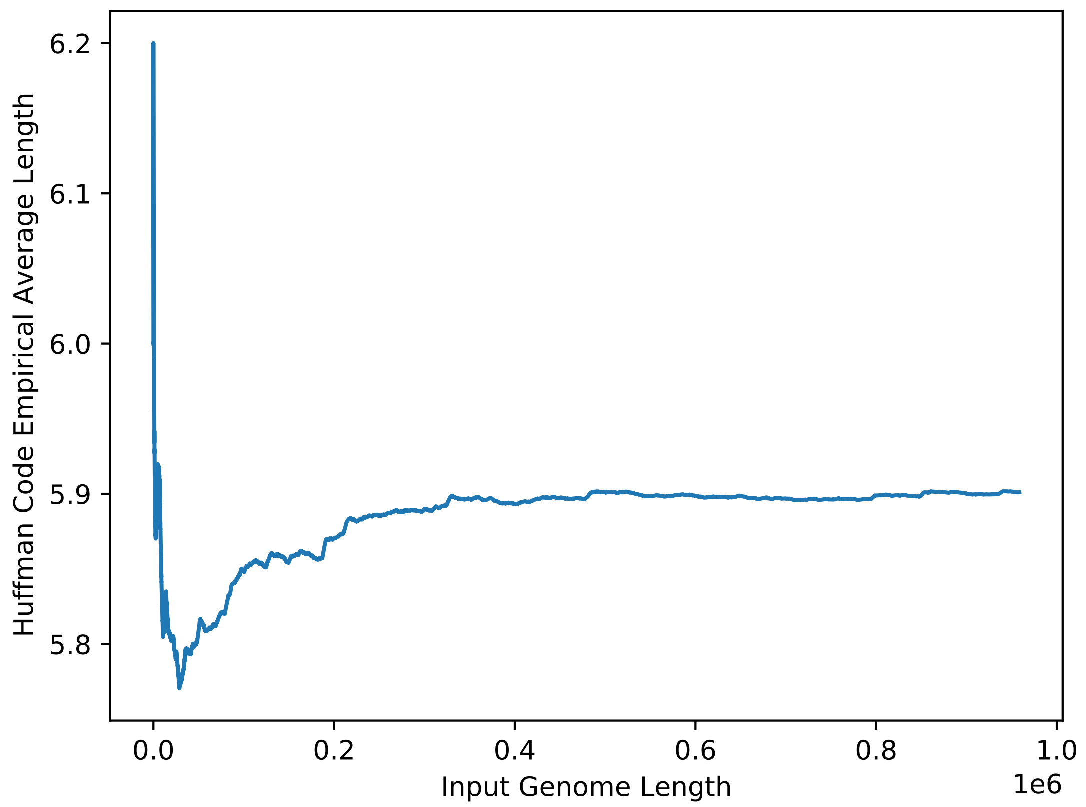

FIGURE 1 – Huffman Code Empirical Average Length

The empirical average length is at first unstable on short genome sequence. On very short sequences, the empirical average length is high (maximum at approximately 6.2) and falls until a minimum of approximately 5.8 for average-length sequence. It stabilises then to the expected average length when the sequence is becoming long enough ($>$ 330 000 bases). This confirms the asymptomatic convergence to the expected average length for long input sequences.

1.8 Question 8

To increase the compression rate of the Huffman code, a good approach would be to increase the grouping of bases to more than 3 bases. With only 4 possible characters 'A', 'C', 'G', 'T' and a 3-base grouping, we have only $4^3 = 64$ possible sequences. We could try to encode sequences of 4 bases for example ($4^4 = 256$ possible sequences). If we try to group bases in sequences of 2 codons (i.e. 6 bases), we would have $4^6 = 4096$ possible sequences. We could even try bigger groupings but it increases the computation time (linearly as a function of the number of possible sequences but exponentially as a function of the number of bases grouped together) and we must check that the different possible sequences do not end up to be equally likely. It also requires to share an increasingly bigger dictionary of frequencies of each group of symbols with the decoding machine (additional overhead).

Another approach could be to adopt some conventions on the encoding and decoding side (less flexibility) based on a priori probabilities of occurrence computed on a database of many genomes. It will be less tailored on the input sequence but could achieve optimal encoding if the a priori distribution matches the one of the sequence that will be given in input. For the encoding of a very long genome, it might actually perform well.

Another approach could be to work with an adaptive data compression variant of Huffman encoding where the Huffman tree structure is incrementally modified on-line after reading each new symbol such as the Knuth-Gallager variant.

1.9 Question 9

When we encode the genome using the Lempel-Ziv algorithm, we obtain a coded genome of length 2697842 and a compression rate of $\frac{7668456}{2697842} = 2.84244$.

1.10 Question 10

When we encode the genome using the LZ77 algorithm (with a sliding window of size 4000), we obtain a coded genome of length 4166648 and a compression rate of $\frac{7668456}{4166648} = 1.84044$.

1.11 Question 11

Combining the LZ77 algorithm and the Huffman algorithm offers two level of compression on different aspects of the data. Such combined approach is exploited in many compression algorithms such as DEFLATE. The LZ77 algorithm is used to compress the repetitive sequences of symbols via, what could be called, pointers towards previously-seen similar sequences of symbols. The complexity and time of encoding can be adapted via the sliding window size. The greater it is, the longer will be the look-up for the longest similar sequences of symbols and the higher will be the complexity and the compression rate. After this first stage, redundancy in sequence of symbols will be greatly reduced. Then, the Huffman algorithm compresses on a symbol per symbol (possibly on a group of symbols per group of symbols) basis based on their frequency (without taking into account correlations between consecutive symbols (resp. group of symbols)). The symbols with higher frequencies will be replaced by sequences of binary symbols of smaller size (in bits). Conversely, the symbols with lower frequencies will be replaced with sequences of binary symbols of greater size (in bits). The redundancy that lies in the number of occurrences of each symbol will thus be optimally reduced.

In practice, we chose to encode the full sequence of 'A', 'C', 'G', 'T' bases with LZ77 and a sufficiently big sliding window of size 1000 because in a genome (but even in general text), repetitions can occur at very distant positions. We could investigate even bigger sliding window at the expense of higher computation time. Then, the result is a sequence of tuples (d, l, c) where they can all have a fixed size representation in binary. We then chose only one of these values to be encoded with Huffman. It ensures no ambiguity in the decoding phase and a smaller dictionary of frequencies to pass on additionally to the coded sequence. If we inspect the values of d, l and c for big window lengths, we can easily see that d vary a lot with few repetitions of the same value. Indeed, for a genome in input, there is about as much chance to find a repetition at any of the d^{th} position in the sliding window. However, l has often small values of 1, 2, 3, ... because most repetitions are small-length repetition in the genome. And finally, c is a sequence of 4 different symbols 'A', 'C, 'G', 'T' that are almost equally likely. Given these observations, we decided to apply Huffman to the values l to obtain a variable length representation of l in binary and benefit the most from Huffman encoding. Finally, to ease the decoding phase, we stacked all the fixed-length tuples (d, c) and then added all variable-length representations of l. We thus obtain an encoded sequence of the form $<d><c><d><c><d><c><l_H><l_H><l_H>$ (where l_H is the Huffman representation of l) for a LZ77 sequence of 3 tuples as input.

1.12 Question 12

When we encode the genome using the LZ77 (with a sliding window of size 4000) and Huffman algorithms combined as described above, we obtain a coded genome of length 2930857 and a compression rate of $\frac{7668456}{2930857} = 2.61646$.

1.13 Question 13

(a)

When we encode the genome using the LZ77 with different sizes l of sliding window, we obtain the following lengths and compression rates for the coded genomes :

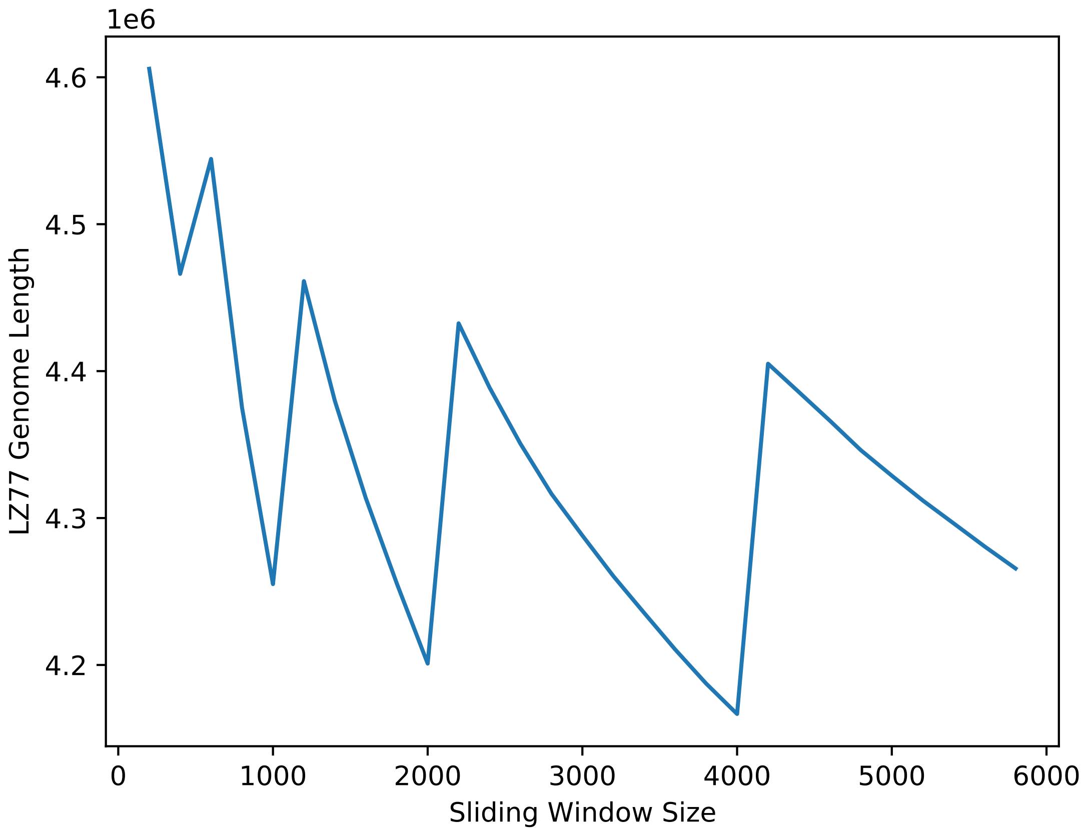

Figure 2 – LZ77 Genome Length

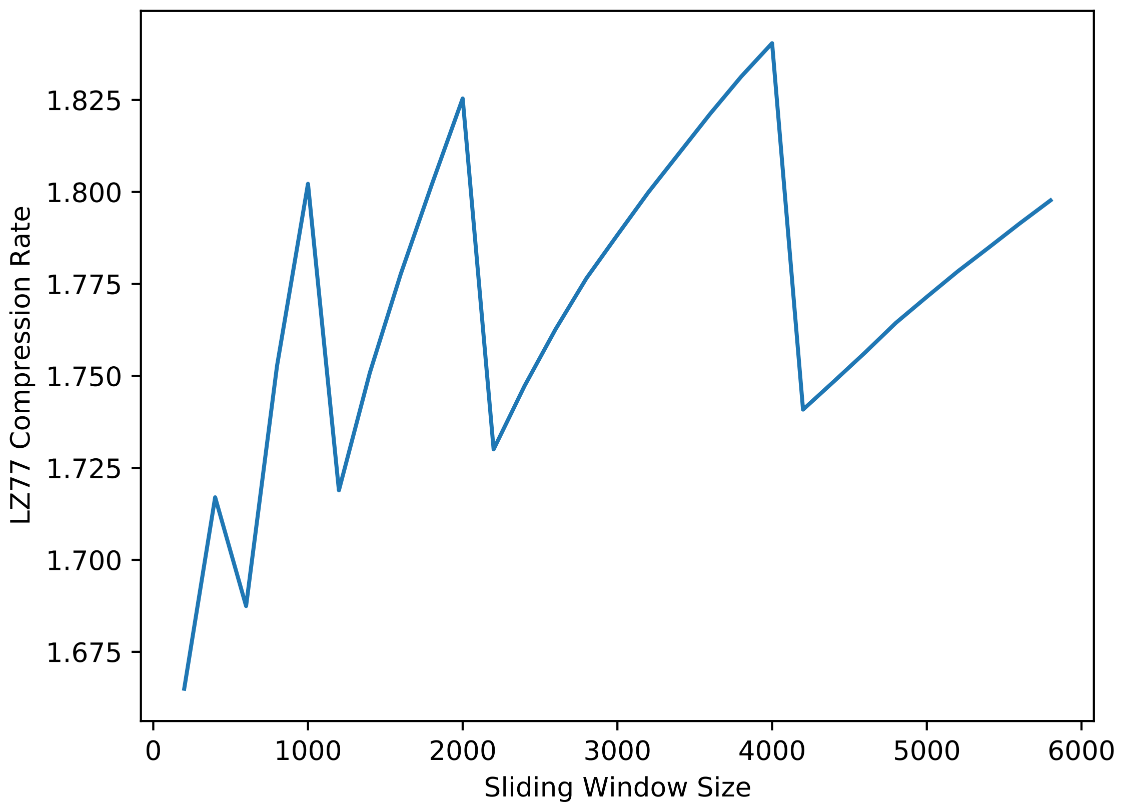

Figure 3 – LZ77 Compression Rate

(b)

When we encode the genome using the LZ77 algorithm with different sizes l of sliding window and the Huffman algorithm combined as described in question 11, we obtain the following lengths and compression rates for the coded genomes :

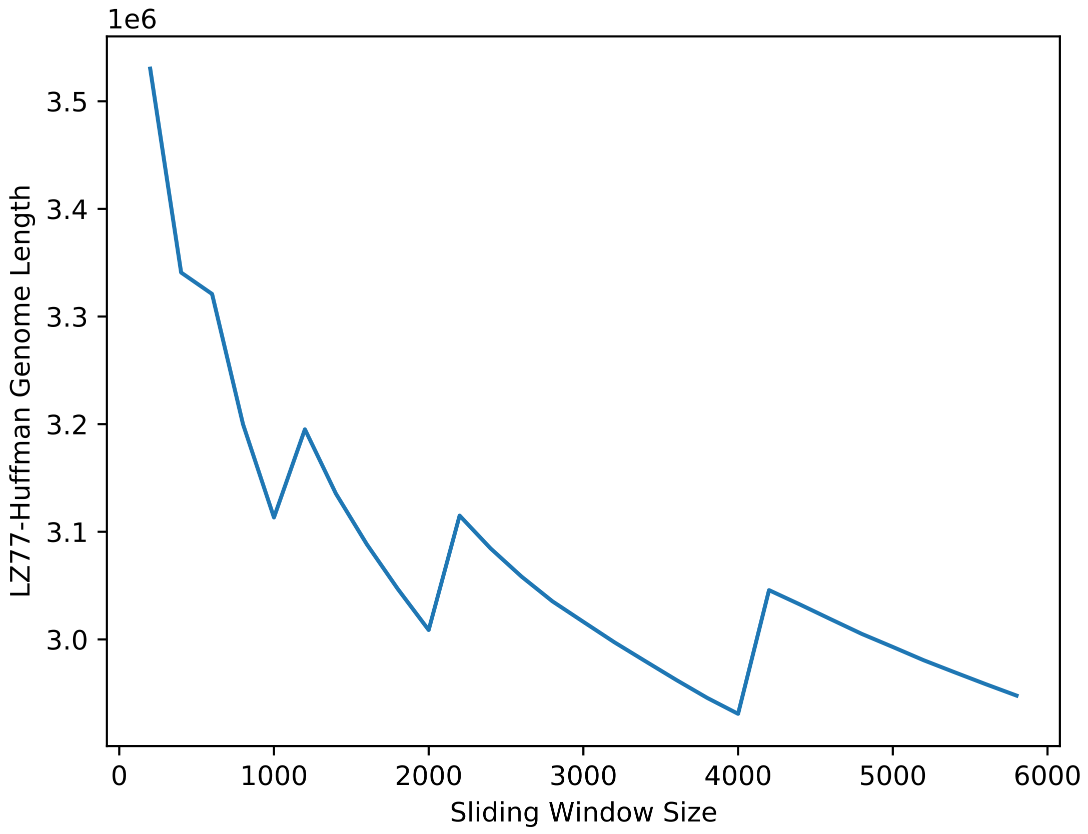

FIGURE 4 – LZ77-Huffman Genome Length

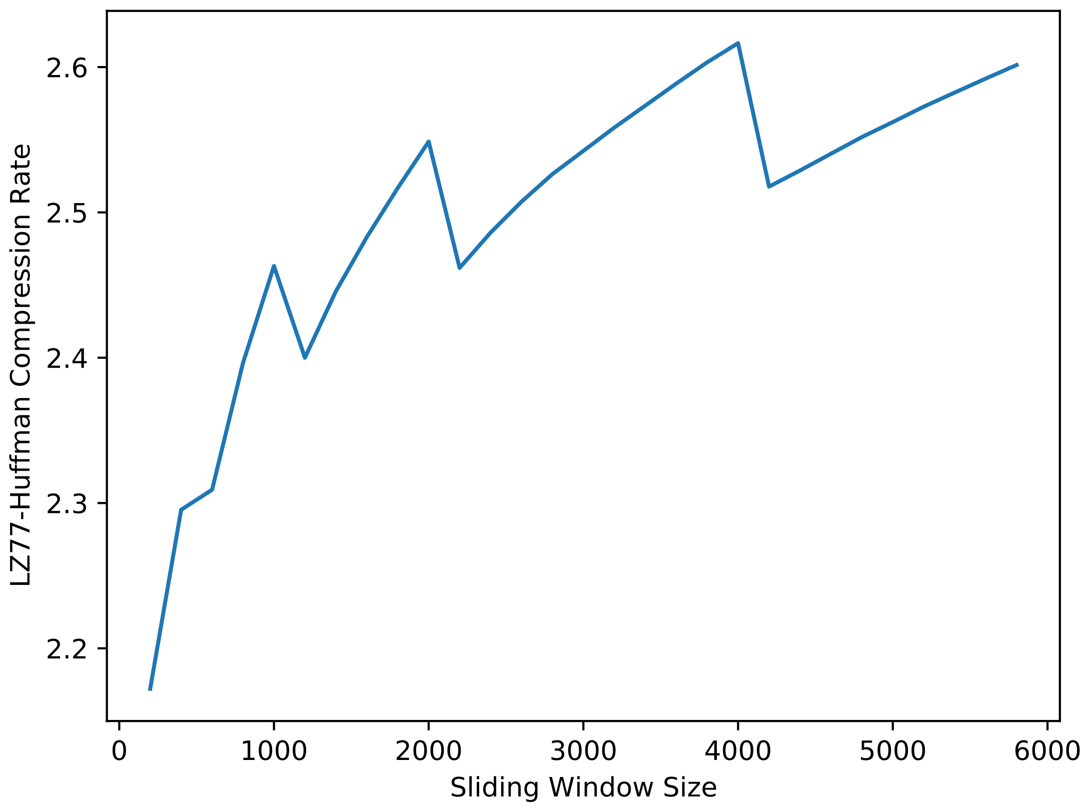

FIGURE 5 – LZ77-Huffman Compression Rate

It is worth noting how the graph of the compression rate ranges within higher values and improves gradually (despite the oscillations) but more significantly as a function of the sliding window size when we make use of Huffman in addition to the LZ77 algorithm.

In comparison to the total length and compression rate obtained using the on-line Lempel-Ziv algorithm, we observe that the Lempel-Ziv algorithm is still performing better with a higher compression rate. Though, we could hope for a better result with LZ77-Huffman if we choose a higher sliding window size.

1.14 Question 14

The best data compression algorithm would probably be a combination of LZ77 and Huffman that looks a bit like the one we tried. The input sequence would indeed benefit from 2 levels of compression on different aspects of the data (as already discussed in Q11), thus reducing 2 different types of redundancy. Moreover, knowing that repetitions occur at long distance in a genome, the LZ77 algorithm could really perform well with a long sliding window (longer than what we tried with our own computers in a limited time). Moreover, our discussion from Q11 on the fact that most (but not all) repetitions tend to be of small length is still valid. We would thus make great use of Huffman to encode the l values coming from LZ77. Additionally, Huffman could be further improved to its adaptive variant : Knuth-Gallager to benefit from a full on-line encoding as LZ77 is already on-line. This would be worth a shot as genomes are very very long sequences that may cause memory overflows if not read in smaller chunks.

However, it is worth noting that for the length of the genome we have (and with the sliding windows we tried), one of the best performing algorithm in terms of compression rate (= 4) and the best in terms of complexity is simply to encode each character 'A', 'C', 'G', 'T' in binary with for example the following dictionary {'A' : '00', 'C' : '01', 'G' : '10', 'T' : '11'} (very small overhead). The best performing algorithm (compression rate = 4.06704) is a plain Huffman where we encode codons but the overhead of passing a dictionary of $4^3 = 64$ entries is larger. Therefore, for small- (or medium-) length sequence, we would advise to simply resort to the one of these two encoding strategies.

2 Channel coding

2.1 Question 15

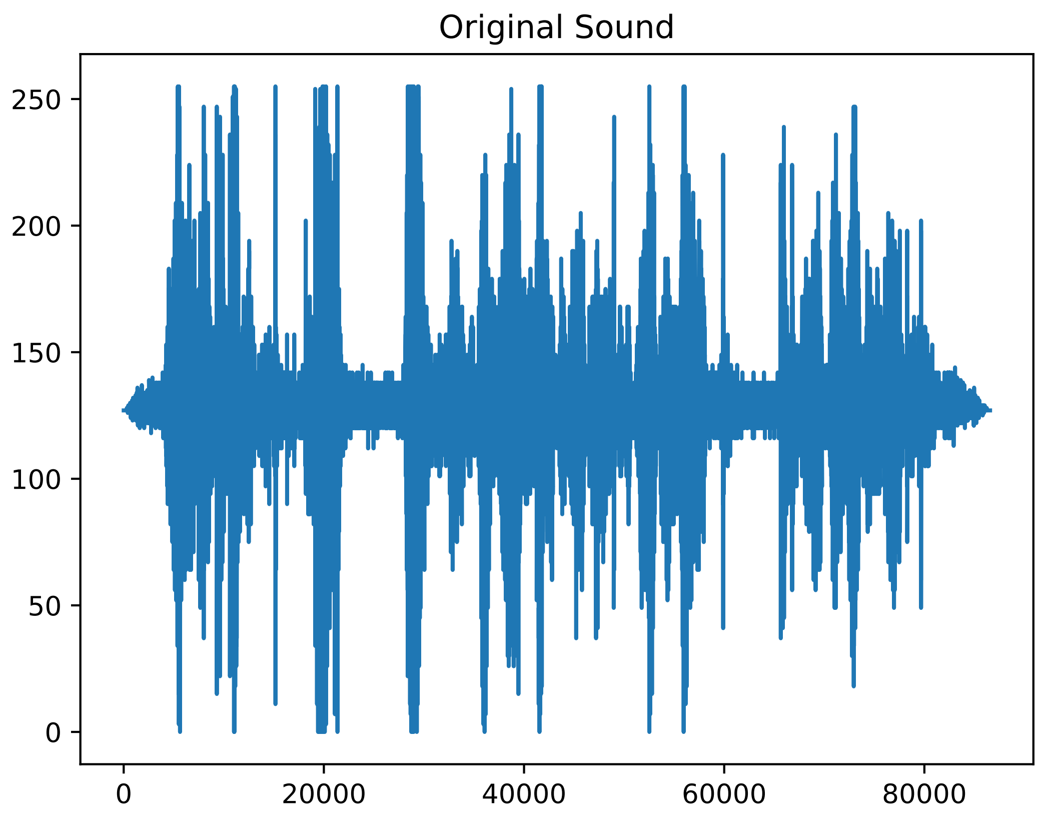

FIGURE 6 – Original Sound

2.2 Question 16

Since the possible values of the signal are between 0 and 255, it seems logical to take 8 bits to encode the sound signal ($2^8 = 256$ distinctive values). However we can see on this plot that only 90 values are used in the signal, which means that we could encode the signal on 7 bits rather than 8. But for 2 reasons we have decided to encode the value on 8 bits anyway. Firstly, we want our encoding to be as general as possible to allow other signals to pass on the channel. Secondly, since we need to add redundancy with Hamming(7,4) code, we would like to stay with a number of bits divisible by 4.

NB : Only these values are used : 0 3 7 11 15 18 22 26 30 34 37 41 45 49 52 56 60 64 67 71 75 79 82 86 90 94 97 101 105 109 112 113 116 117 118 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 144 145 149 153 157 160 164 168 172 175 179 183 187 190 194 198 202 205 209 213 217 220 224 228 232 236 239 243 247 251 254 255.

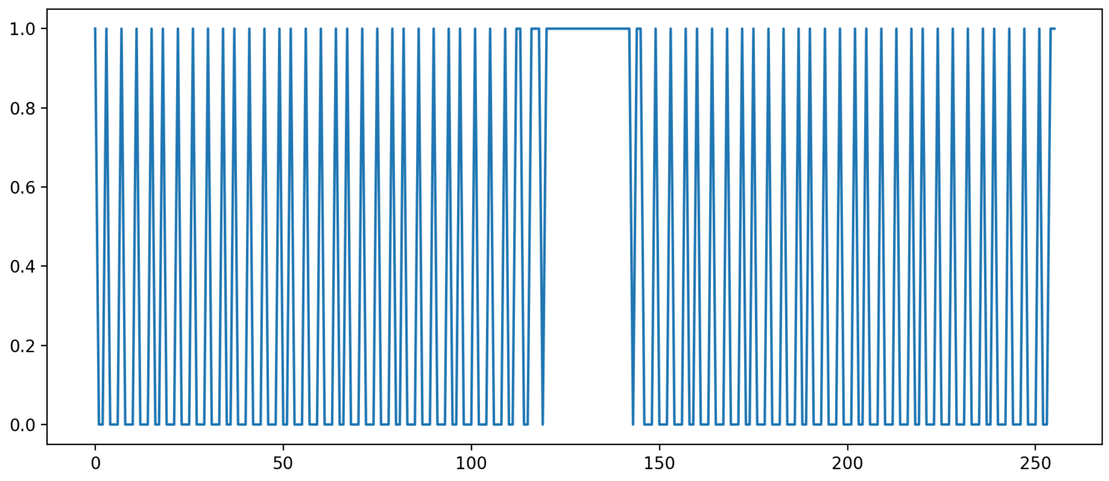

FIGURE 7 – Distinctive Values of the Signal

2.3 Question 17

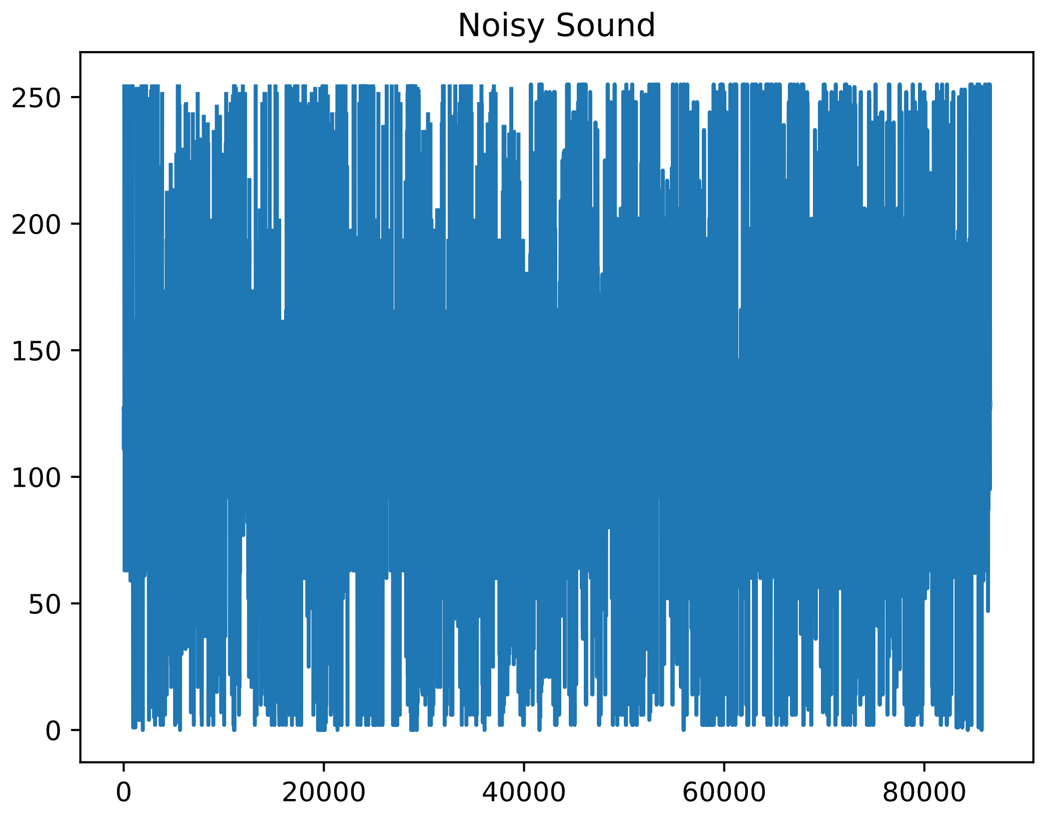

FIGURE 8 – Noisy Sound

We can clearly see that we have a lot of noise. But this is expected since we have a probability of error of 0.01, so 1 bit over 100 should be flipped on average. Furthermore, we encoded the value on 8 bits, so this means that more or less 1 over 12.5 values from the original sound is altered. And on a scale of 692520 bits, we have around 6925 bits that are wrongly decoded (empirically, it amounts to 6623 wrongly decoded values in the range $[0, 255]$).

When we listen to the noisy decoded signal, we hear a lot of noise (like crackelings) but the message can still be understood. It is quite interesting to note how well human speech remains understandable for a human hear despite the noise. It is also some sort of natural channel decoding that evolved throughout history.

2.4 Question 18

See the code.

2.5 Question 19

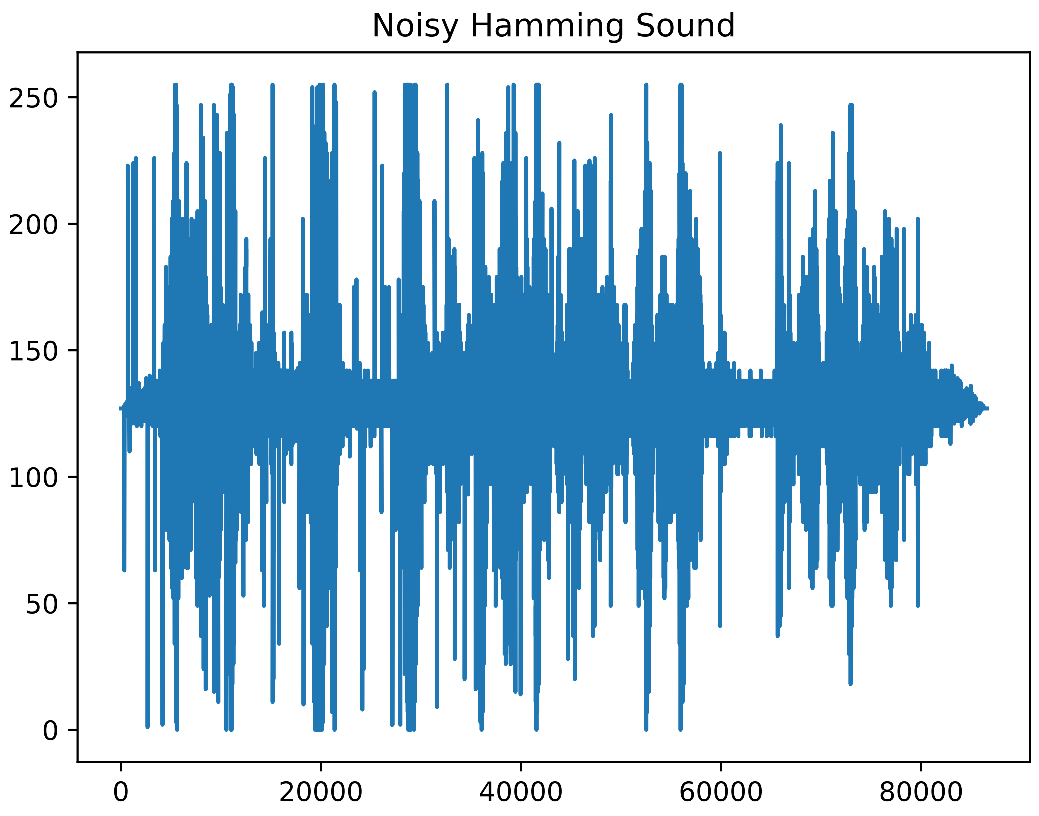

Figure 9 – Noisy Hamming Decoded Sound

We can clearly see the effect of the Hamming code and that adding redundancy really helps to get back a signal close to the original sound and thus yield a more pleasing audio result.

With Hamming(7,4), we have the capacity to correct maximum 1 error if it happens only to 1 bit. To encode a value in the range $[0, 255]$, we need 8 bits. With Hamming(7, 4), these 8 bits of effective information amount to 14 bits with redundancy. To have a wrong decoded value, we have to get at least two errors in one codeword. Having a wrong decoded value in the range $[0, 255]$ happened empirically 213 times only.

The decoding procedure is described in the following steps. Run through the encoded sound by considering each time 7 bits by 7 bits :

1. Since we know that the received message was encoded with an Hamming $(7, 4)$ code, we know that the first 4 bits correspond to the signal(s_i), and that the last 3 are the parity bits.
2. Then, in order to compute the syndrom, we will evaluate the 3 expected parity bits of the message from the signal :

$$\begin{aligned} expected\_p_1 &= (s_1 + s_2 + s_3) \mod 2 \\ expected\_p_2 &= (s_2 + s_3 + s_4) \mod 2 \\ expected\_p_3 &= (s_1 + s_3 + s_4) \mod 2 \end{aligned}$$

Once the expected parity bits obtained, we can compute the syndrom as follows :

$$\begin{aligned} syndrom_1 &= (p_1 + expected\_p_1) \mod 2 \\ syndrom_2 &= (p_2 + expected\_p_3) \mod 2 \\ syndrom_3 &= (p_2 + expected\_p_3) \mod 2 \end{aligned}$$

As it is coded on three bits, there are 8 possible different error patterns represented in the table below. These values are contained in a dictionary in the implemented function where the keys are the syndroms and the values their corresponding errors.

Syndrom	Corresponding error
000	0000000
001	0000001
010	0000010
011	0001000
100	0000100
101	1000000
110	0100000
111	0010000

Table 1 – Syndrom and Corresponding Error

3. Now that the syndrom is computed, the error can easily be retrieved thanks to the dictionary created previously. Therefore, the message can be corrected by adding bit by bit the signal and error. (Note that both of them are coded on 7 bits) Then, we can only keep the first four bit of the last step and hope that the corrected message obtained is the good one.

   Nb : However, in the code, since we are only focusing on the error on the first four bits, we simplified the procedure a little bit and only use the syndrom to change the first four bits.

2.6 Question 20

To improve the communication rate, we could make use of another Hamming code. With m parity bits, Hamming$(2^m - 1, 2^m - m - 1)$ offers a communication rate of $\frac{2^m-m-1}{2^m-1}$ that will thus converge towards 1 for increasing values of m.

To reduce the loss of information, we should make use of a more efficient forward error correction code. It however comes at the cost of a decoding scheme of higher complexity (eg : convolution codes decoded with the Viterbi algorithm) or of lower communication rate (eg : repetition code). For example, repetition codes work by repeating each effective bit of information n times. The greater n is, the lower the communication rate $\frac{1}{n}$ is but the lower the bit error rate is. For instance if n = 3, we have a $BER = p^3 + C_3^2 p^2(1-p) = 0.028$, quite smaller than the BER of Hamming(7, 4) of 0.07. Nevertheless, Hamming still has a higher communication rate of $\frac{4}{7}$ than the repetition code $\frac{1}{3}$

So, depending on the situation we would use different method. If the focus is on losing no information we would prefer repetition code than Hamming. But if the focus it doesn't matter to have few losses but we want a high communication rate, we would prefer Hamming than repetition code.

Another thing that we can do in order to both increase the communication rate and reduce the loss of information is to decrease the expected length of the signal. In fact, we used 8 bits to encode the value. However, since we pointed out in Q19 that the signal only use 90 values, we could easily use 7 bits instead to encode a value and have thus a shorter length.[1] As expected, for question 17, we notice that the number of error is much smaller (around 6059 errors rather than 6925). And with the Hamming Code, we expected to see the number of errors drop too but it did not drop much.

In order to decrease the expected length, compression algorithms could also be used as a preprocessing step before channel encoding algorithms. As we can see on Figure 10, the number of occurrences of each value is not balanced. We can clearly see that some values are much more used than others so we could make use of Huffman in order to decrease the length of the signal and therefore decrease both the loss of information and the communication rate.

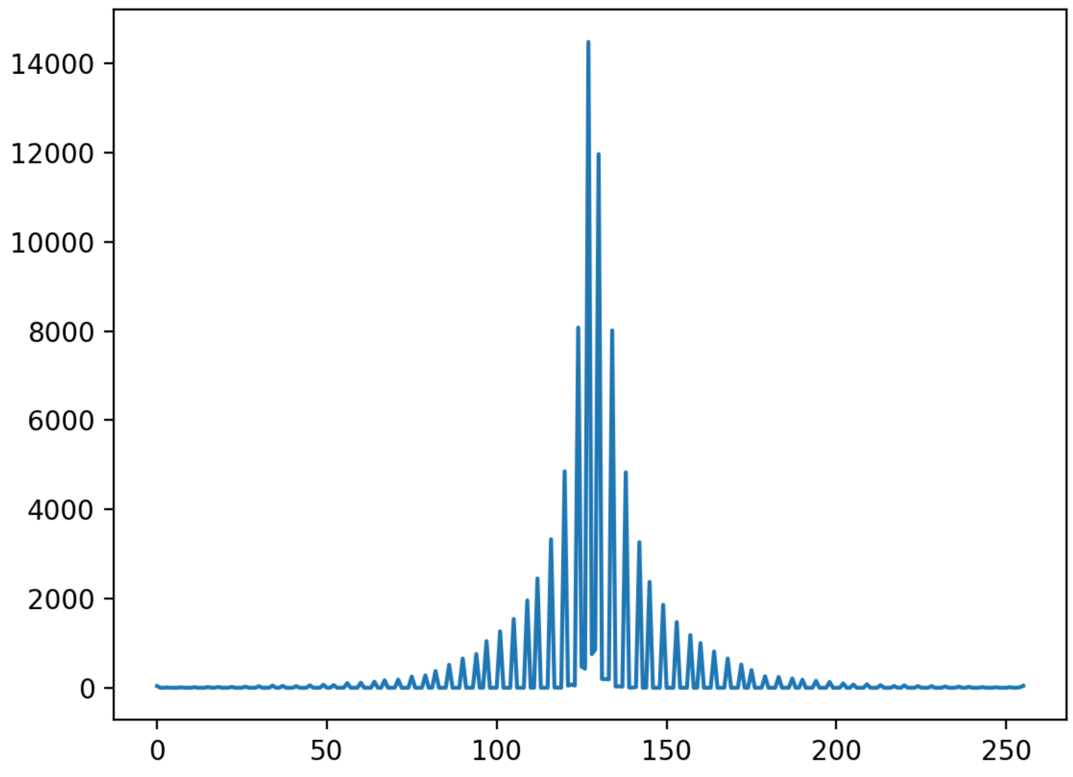

FIGURE 10 – Occurrences of value used

1. We implemented this version in appendix in the code