Up to now, we have looked at codes for a set of symbols in an alphabet. We have also looked at the specific case that the alphabet is a set of integers. We will now study a few compression techniques in which we will code both symbols as well as integers.

Move-to-front algorithm

Version 1

Suppose we are given a sequence of symbols as before, with each symbol coming from an alphabet $\{A_1, \ldots, A_N\}$. The move-to-front algorithm maintains a list of the N symbols in the alphabet. For each symbol in the sequence, it encodes the position of this symbol in the list, and then modifies the list by moving the symbol to the front of the list.

Example

Suppose $N = 6$. Initialize the list of symbols to be $[A_1, A_2, A_3, A_4, A_5, A_6]$.

Suppose the example data sequence to be encoded starts off $A_3A_4A_1A_2A_2A_6A_2 \ldots$. The move-to-front algorithm does the following:

encode	updated list of symbols
	$[A_1, A_2, A_3, A_4, A_5, A_6]$
$C(3)$	$[A_3, A_1, A_2, A_4, A_5, A_6]$
$C(4)$	$[A_4, A_3, A_1, A_2, A_5, A_6]$
$C(3)$	$[A_1, A_4, A_3, A_2, A_5, A_6]$
$C(4)$	$[A_2, A_1, A_4, A_3, A_5, A_6]$
$C(1)$	$[A_2, A_1, A_4, A_3, A_5, A_6]$
$C(6)$	$[A_6, A_2, A_1, A_4, A_3, A_5]$
$C(2)$	$[A_2, A_6, A_1, A_4, A_3, A_5]$
:	:

Thus, the encoder writes down or sends the sequence of codewords,

$$C(3)\ C(4)\ C(3)\ C(4)\ C(1)\ C(6)\ C(2)\ \ldots$$

How can a decoder recover the original data from the sequence of codewords? It does so by mimicking the encoder. It maintains a list of the N symbols, initializing the list in the same order as the encoder initializes its list. Then, for each codeword $C(i)$ that it reads, it looks at the symbol at position i in its current list of the N symbols. This is the symbol that has been encoded. To continue, it moves the symbol to the front of its list. [Verify for yourself that the decoding algorithm recovers the original sequence for the example above. If you don't understand, come and see me.]

At first glance, it may not be obvious why move-to-front should compress the sequence. The key observation is that *symbols that occur with higher probability tend to be nearer the front of the list*. Each time the symbol occurs, it is put at the front of the list. If the symbol occurs frequently, then it won't drop too far back in the list before it occurs again. By contrast, a symbol that occurs rarely tends to fall relatively far back in the list between occurences, because other symbols are

moved in front of it. So, by using shorter codewords $C(i)$ for positions i near the front of the list (i.e. smaller i), i.e.

$$\lambda_1 \leq \lambda_2 \leq \lambda_3 \leq \ldots$$

we tend to use shorter codewords for frequently occuring symbols. But this is what we need to do to achieve compression. So, in this sense, move-to-front should work.

Best case performance of move-to-front

Move to front performs best when we have a symbol that is repeated over and over. e.g.

$$\ldots A_3 A_3 A_3 A_3 A_3 A_3 \ldots$$

The first time A_3 occurs in this sequence, we need to encode its position in the list, but each subsequent time, it is already at the front of the list and so it is cheap to encode. At least, it is cheap to encode if $C(1)$ is short.

In practice, we do not expect to have such best case sequences. But the example suggests a general way in which move-to-front compresses well. If the sequence has certain regions in which particular symbols occur more frequently, the algorithm takes advantage of this: it uses shorter codewords since the more frequently occuring symbol tends to be near the front of the list.

Such variations in local frequency, i.e. *clustering*, does arise is real world data. For example, if you open a randomly chosen book from the library and you see the word "fish," then it is more likely that the word "fish" will appear again in that particular book (or chapter, or paragraph) than if you chose another random book (or chapter, or paragraph).

Worst case performance

What about the worst case performance? You might expect that the worst case for move-to-front would occur for the sequence:

$$A_N A_{N-1} A_{N-2} \ldots A_1 A_N A_{N-1} A_{N-2} \ldots A_1 A_N A_{N-1} A_{N-2} \ldots A_1$$

The reasoning is that the symbol to be coded next would always lie at the end of the list. Hence the average codeword length would be $\lambda(N)$, which is maximal. (Note, however, that this worst case requires that each symbol occur an equal number of times, and hence be "equally likely". When not all symbols are equally likely, this worst case arrangement of the symbols is impossible.)

Version 2

To analyze the worst-case performance of move-to-front, we will use a slightly different version of the algorithm. As before, suppose we have a prefix code C for the positive integer positions in a list of symbols. We also consider a prefix code C^* for the symbols in the alphabet. These two codes have nothing to do with each other.

In the new version of the algorithm, the encoder starts with an *empty* list. For each symbol A in the sequence, it checks whether that symbol is in the list. If so, and if the symbol is at position j, then the codeword $C(j)$ is sent and the symbol is moved to the front. If the symbol is not in

the list (i.e. it has not been seen before in the sequence), and if there are k elements in the list currently ($k \geq 0$) then the codeword $C(k+1)$ is sent – that is, the symbol is treated as if it were last in the list. This new symbol is then inserted at the front of the list (i.e. moved to front), and the code for the symbol $C^*(A)$ is sent.

The decoder does the same thing. It begins with an empty list. Then, at any time there are k symbols in the list, it reads the next codeword which is some $C(i)$. If $i = k+1$, then this means the next symbol in the sequence is not in the current list; so it interprets the next bits as the codeword for this new symbol $C^*(k+1)$. It then inserts this new symbol at the front of the list. But if $i \leq k$, then the symbol must be in the list, namely at position i in the list, and so it looks up this symbol in the list and moves it to the front.

Example

Suppose the example sequence to be encoded starts off $A_3A_4A_1A_2A_2A_6A_2\ldots$.
The move-to-front algorithm does the following. The list is initially empty.

encode	updated list
	empty
$C(1)C^*(A_3)$	$[A_3]$
$C(2)C^*(A_4)$	$[A_4, A_3]$
$C(3)C^*(A_1)$	$[A_1, A_4, A_3]$
$C(4)C^*(A_2)$	$[A_2, A_1, A_4, A_3]$
$C(1)$	$[A_2, A_1, A_4, A_3]$
$C(5)C^*(A_6)$	$[A_6, A_2, A_1, A_4, A_3]$
$C(2)$	$[A_2, A_6, A_1, A_4, A_3]$
:	:

You should make sure that you understand how the decoder works, namely, make sure that you can decode the sequence and get back the original example.

$$C(1)C^*(A_3)C(2)C^*(A_4)C(3)C^*(A_1)C(4)C^*(A_2)C(1)C(5)C^*(A_6)C(2)$$

One advantage of this second version of the move-to-front algorithm is that the list is much shorter in the case that the number of symbols N in our alphabet is large. For example, suppose our alphabet is the set of words in the English dictionary. Do we really want to create a list of these words and start moving elements to the front? No.

Worse case analysis of move-to-front (version 2)

Let us calculate an upper bound (worst case) on the total number of bits used by the C code, i.e. the codewords that indicate positions in the move-to-front list. We ignore the number of bits used by C^* code which is the code used for the symbols themselves; the issue of how to encode the individual symbols is independent of how to encode based on the frequency/pattern of occurences. Moreover, the C^* code is used only once for each symbol that occurs in the sequence.

Consider a sequence of n symbols which we wish to encode. Let the symbols that occur in this sequence be denoted $\{A_1, A_2, \ldots, A_N\}$, where $N \leq n$ by definition. Let $n_i \geq 1$ be the number of

occurences of symbol A_i. Let the times when A_i occurs be denoted $t_1^i, t_2^i, \ldots, t_{n_i}^i$. One way to visualize these t's is to use an $N \times n$ matrix with values in 0 and 1. In row i and column m, we have a 1 if and only if A_i occured at time m. The number of 1's in row i is thus n_i, and there is a unique 1 in each column.

For the example sequence $A_3A_4A_1A_2A_2A_6A_2\ldots$ and assuming $N = 6$, the matrix of occurance positions t_m^i would be

$$\begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 & \ldots \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 & \ldots \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & \ldots \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & \ldots \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \ldots \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & \ldots \end{pmatrix}$$

where i is row and m is column.

Let λ_{tot}^i be the number of bits used to encode the positions of these n_i occurances of A_i. There are two key observations.

- The first time symbol A_i occurs, the list can have at most $t_1^i - 1$ elements, and so the position encoded by move-to-front is at most t_1^i. This requires at most $\lambda(t_1^i)$ bits.
- From its k^{th} occurence to its $k+1^{st}$ occurence, a symbol A_i can fall at most to position $t_{k+1}^i - t_k^i$ in the list. Thus, to encode this position requires at most $\lambda(t_{k+1}^i - t_k^i)$ bits.

Hence,

$$\lambda_{tot}^i \leq \lambda(t_1^i) + \lambda(t_2^i - t_1^i) + \lambda(t_3^i - t_2^i) + \cdots + \lambda(t_{n_i}^i - t_{n_{i-1}}^i)$$

We next simplify this inequality using Jensen's inequality. Suppose we have an upper bound on the function $\lambda(\ )$, namely a continuous non-decreasing convex function $f(x)$ such that

$$\lambda(j) \ \leq \ f(j)$$

where j is a positive integer. For example, recalling the Elias code with codeword lengths

$$\lambda(j) = 2\lfloor \log j \rfloor + 1$$

we could take

$$f(x) \equiv 2\log(x) + 1 \ .$$

We combine the two inequalities above to get a (slightly higher) upper bound for λ_{tot}^i

$$\lambda_{tot}^i \ \leq \ f(t_1^i) + f(t_2^i - t_1^i) + f(t_3^i - t_2^i) + \cdots + f(t_{n_i}^i - t_{n_{i-1}}^i)$$

Next, we multiply by $\frac{n_i}{n_i}$ and apply Jensen's inequality. [Note that Jensen's inequality was proven in class for $\log(x)$ but exactly the same proof holds for any convex function $f(x)$. In particular, see the Appendix of lecture 5.]

$$\begin{aligned} \lambda_{tot}^i \quad &\leq \quad n_i \ \frac{1}{n_i} \ (f(t_1^i) + f(t_2^i - t_1^i) + f(t_3^i - t_2^i) + \cdots + f(t_{n_i}^i - t_{n_{i-1}}^i)) \\ &\leq \quad n_i \ f(\ \frac{1}{n_i} \ (t_1^i + (t_2^i - t_1^i) + (t_3^i - t_2^i) + \cdots + (t_{n_i}^i - t_{n_{i-1}}^i))) \\ &= \quad n_i \ f(\frac{t_{n_i}^i}{n_i}) \qquad \text{by ''telescoping''} \\ &\leq \quad n_i \ f(\frac{n}{n_i}) \qquad \text{since} \ \ t_{n_i}^i \leq n \ \ \text{and} \ f() \ \text{is non decreasing} \end{aligned}$$

To get an upper bound on the average codeword length (for the code C), we sum up the λ^i_{tot} over all i, then divide by the number of symbols n in the sequence,

$$\overline{\lambda} = \frac{1}{n} \sum_{i=1}^{N} \lambda^i_{tot} \quad \leq \quad \sum_{i=1}^{N} \frac{n_i}{n} f(\frac{n}{n_i}) \tag{1}$$

That is, for all sequences with alphabet A_i, $i = 1, .., N$ and with frequencies n_i, the average codeword length will be at most this much.

For example, consider the Elias1 code. We can bound $\lambda(\ )$ by $f(x) = 2\log x + 1$. Thus, if we use the Elias1 code, then the average code length satisfies:

$$\overline{\lambda} \quad \leq \quad \sum_{i=1}^{N} \frac{n_i}{n} f(\frac{n}{n_i}) \quad = \quad \sum_{i=1}^{N} \frac{n_i}{n} \ (\ 2\log(\frac{n}{n_i}) + 1) \ = \ 1 + 2\sum_{i=1}^{N} \frac{n_i}{n} \ \log(\frac{n}{n_i}) \tag{2}$$

Returning to Eq. (1), notice that, for $n \gg N$, the proportion $\frac{n_i}{n}$ will converge to the probability that symbol A_i occurs at any particular element in the sequence. Thus, as $n \ \rightarrow \ \infty$, we have

$$\sum_{i=1}^{N} \frac{n_i}{n} \ f(\frac{n}{n_i}) \ \rightarrow \ \sum_{i=1}^{N} \ p(A_i) \ f(\frac{1}{p(A_i)}) \tag{3}$$

In particular, if we use the Elias1 code, then as $n \rightarrow \infty$ (see Eq. (2)), we have

$$\overline{\lambda} \leq 2H + 1$$

where H is entropy that is defined by the probabilities $p(A_i)$.

At first glance, you might think that this is a very weak upper bound. You would hope your coding scheme would give an average code length that is near the entropy, not twice the entropy! However keep in mind that Eq. (1) defines a bound on the worst case, given the frequencies $n_1, n_2, \ldots, n_N$. Move-to-front can perform much better than the bound of Eq. (1) e.g. if the symbols cluster together, that is, if the occurence of A_i in any position makes it more likely that A_i will occur at a nearby position. Even if such clustering doesn't occur, the bound is arguably still interesting: it holds without knowing in advance the probability of the symbols occuring!

Finally, note that if we use the Elias2 code, then using exactly the same argument, you can show that

$$\overline{\lambda} \leq H + 2\log(H + 1) + 1.$$

For large H, this average code length is really not much worse than H. This is especially impressive, given that it is the worst case performance on the average code length. In practice, there will be clustering, which will lead to be performance.