CSCI-GA.1170-001/002 Fundamental Algorithms November 20, 2012

On-Line Prefix-Free Codes

Yevgeniy Dodis *Supplementary Notes*

Recall, an encoding E is called prefix-free, if for any distinct messages $M_1 \neq M_2$, we have that $E(M_1)$ is not a prefix of $E(M_2)$. Below we show a very simple *on-line* prefix-free encoding, where the message (resp. encoding of the message) can be encoded (resp. decoded) on the fly. In particular, the encoding/decoding algorithms do not know the message length in advance, and terminate only when explicitly encoding/decoding the special "end-of-file" (EOF) symbol. Moreover, the encoding/decoding have several extremely nice "locality" properties, as explained in the text below.

1 The SOLE Encoding

We assume that the input stream M consists of blocks of $b \geq 2\log_2 n + 1$ bits, and that n is odd (we can pad with an additional EOF block otherwise). Under these conditions, our algorithm will output an encoding of $n+2$ blocks. For each block, the algorithm uses $O(1)$ arithmetic operations on b-bit integers.

Let $B = 2^b$ be the alphabet of a block, and we write $[B]$ to denote the range $\{0, \ldots, B-1\}$. Adding a special end-of-file symbol (EOF), we obtain an alphabet of size $B+1$. Our goal is to encode $n+1$ letters from this alphabet (including the final EOF) using $n+2$ blocks of b bits.

The algorithm is best illustrated by Figure 1. Intuitively, it is simplest to view the algorithm as two separate passes through the stream.

In *Pass 1*, we consider pairs of elements on positions $(2i+1, 2i+2)$, for $i \geq 0$. Grouped together, the elements form a number in range $\left[(B+1)^2\right]$: namely, we view the two elements $x, y \in [B+1]$ as one element $z = y \cdot (B+1) + x \in [(B+1)^2]$. We decompose this number into two values: one in range $\left[B-4i\right]$, and one in range $\left[B+4i+4\right]$ (see next paragraph). The smaller range is placed on odd positions, while the larger one on even positions, hence the name "Short-Odd Long-Even (SOLE)[1] Encoding."

The latter decomposition is simply a division by $B-4i$, using the remainder as the first block and the quotient as the next one. Namely, if $z = y' \cdot (B-4i) + x'$, then $x' \in [B-4i]$ and, as we show below, $y' \in [B+4i+4]$. Namely, the quotient $0 \leq y' < B+4i+4$. To show this, we only need to show the following inequality:

$$(B+1)^2 \leq (B-4i)(B+4i+4) \Longleftrightarrow B \geq 2(2i+1)^2 - \frac{3}{2}$$

which follows from the fact that $2i+1 \leq n$ and $B \geq 2n^2$.

In *Pass 2*, we regroup the elements, considering pairs on positions $(2i, 2i+1)$ for $i \geq 1$. These elements come from range $\left[B+4i\right]$ and $\left[B-4i\right]$, respectively. Since $(B+4i)(B-4i) < B^2$, we can group them together as a number in range $[B^2]$. This uses a multiplication by $B+4i$, as follows: if the blocks are $x \in [B+4i]$ and $y \in [B-4i]$, we set $z = y \cdot (B+4i) + x$, so that $z < (B-4i)(B+4i) < B^2$ and $z \in [B^2]$. The latter value z we can simply write in binary as $2b$ bits. Hence, we have obtained $2b$ bits (a "double-block"), which we output immediately.

It is clear that these two conceptual passes can be implemented as a single pass in an online streaming algorithm. We simply need to remember the state of each pass (at most two integers each). The decoding

[1] Coincidentally, one meaning of the word "sole" is "exclusive, not shared with others".

Block Number	1	2	3	4	5	6	...	$n = 2i+1$	$2i+2$	$2i+3$
Input Alphabet	$[B]$	$[B]$	$[B]$	$[B]$	$[B]$	$[B]$	...	$[B]$	$\{\text{EOF}\}$	$\{0\}$
Size with EOF	$B+1$	$B+1$	$B+1$	$B+1$	$B+1$	$B+1$	...	$B+1$	$B+1$	$B-4i-4$
Pass 1	B	$B+4$	$B-4$	$B+8$	$B-8$	$B+12$	...	$B-4i$	$B+4i+4$	$B-4i-4$
Regroup	B	$B+4$	$B-4$	$B+8$	$B-8$	$B+12$	...	$B-4i$	$B+4i+4$	$B-4i-4$
Pass 2	B	B	B	B	B	B	...	B	B	B

Figure 1: Short-Odd Long-Even (SOLE) Encoding. Assumes n is odd and $B \geq 2n^2$.

Block Number	1	2	3	4	5
Input Alphabet	[32]	[32]	[32]	$\{\text{EOF}\}$	$\{0\}$
Size with EOF	33	33	33	33	24
Pass 1	32	36	28	40	24
Regroup	32	36	28	40	24
Pass 2	32	32	32	32	32

Figure 2: SOLE Encoding for $n = 3$ and $B = 32$. Note how, for example, $36 \cdot 28 < 32^2 < 33^2 < 28 \cdot 40$.

algorithm is the straightforward inversion of this process, implementing Figure 1 bottom-up. One can immediately observe the following locality property:

Property 1 *Output blocks $2i$ and $2i+1$ can be computed from four input blocks $2i-1, 2i, 2i+1, 2i+2$. Similarly, input blocks $2i+1$ and $2i+2$ can be decoded from four output blocks $2i, 2i+1, 2i+2, 2i+3$.*

TERMINATION. An important component of the algorithm that we have not described is the termination behavior, once EOF is received. We assumed that n was odd, i.e. EOF appears at some even position $n+1 = 2i+2$. The final element after Pass 1 is in range $\left[B+4i+4\right]$. We artificially insert a zero value in range $\left[B-4i-4\right]$ into the stream output by Pass 1. After regrouping and Pass 2, this completes a double-block at positions $2i+2$ and $2i+3$, which is output.

When the decoding algorithm has decoded the EOF symbol, it stops immediately. Thus, we need to argue that the decoding stops before reading past the end of the encoded file. This follows from Property 1, as the last block needed in the decoding of EOF is $2i+3$.

1.1 Small Example

Assume the number of blocks $n = 3$ and $B = 32$, so each block is a 5-bit number in the range $[32] = \{0, \ldots, 31\}$ and the last index $i = 1$. This is OK as n is odd and $32 > 2 \cdot 3^2 = 18$. Figure 2 denotes the specialization of Figure 1 to this concrete setting. In contrast, Figure 3 denotes the actual run of the encoding (and recoding, if done in reverse) on 3-block input $(5, 7, 31)$. Please refer to these two Figures to follow the calculations below.

1.1.1 Encoding

Say, the 3 input blocks are $5, 7, 31$. First, we attach the EOF, which is encoded as 32. We also add the last 0 which is viewed as being in range $B-4i-4 = 32-4-4 = 24$. Thus, we get a modified stream of 5 blocks $5, 7, 31, 32, 0$, where $5, 7, 31, 32 \in [33]$ and $0 \in [24]$. We get two "double blocks" $(5, 7)$ and $(12, 31)$. Let us process them one-by-one, as a streaming algorithm would do.

Block Number	1	2	3	4	5
Input Alphabet	[32]	[32]	[32]	{EOF}	{0}
Actual Input	$5 \in [32]$	$7 \in [32]$	$31 \in [32]$	EOF	n/a
With EOF	$5 \in [33]$	$7 \in [33]$	$31 \in [33]$	$32 \in [33]$	$0 \in [24]$
Pass 1	$12 \in [32]$	$7 \in [36]$	$23 \in [28]$	$38 \in [40]$	$0 \in [24]$
Regroup	$12 \in [32]$	$7 \in [36]$	$23 \in [28]$	$38 \in [40]$	$0 \in [24]$
Pass 2	$12 \in [32]$	$3 \in [32]$	$26 \in [32]$	$6 \in [32]$	$1 \in [32]$

Figure 3: Actual run for $n = 3$, $B = 32$ and $M = (5, 7, 31)$.

First Double Block. We need to encode $(5, 7) \in [33] \times [33]$ as two numbers in the range $[32] \times [36]$. First, from $x = 5$ and $y = 7$, we obtain a large number $z = 7 \cdot 33 + 5 = 236$. Then we uniquely decompose 236 modulo 32: $236 = 7 \cdot 32 + 12$. Thus, our first double block is $(12, 7) \in [32] \times [36]$.

Second Double Block. We need to encode $(31, 32) \in [33] \times [33]$ as two numbers in the range $[28] \times [40]$. First, from $x = 31$ and $y = 32$, we obtain a large number $z = 32 \cdot 33 + 31 = 1087$. Luckily, as we expected, $1087 < 28 \cdot 40 = 1120$, so we can uniquely decompose 1087 modulo 28 as: $1087 = 38 \cdot 28 + 23$. Thus, our second double block is $(23, 38) \in [28] \times [40]$.

Output of Pass 1 and Regrouping. Thus, after Pass 1, we transformed our modified stream $(5, 7, 31, 32, 0) \in [33] \times [33] \times [33] \times [33] \times [24]$ into $(12, 7, 23, 38, 0) \in [32] \times [36] \times [28] \times [40] \times [24]$. We now regroup it into a singleton element $12 \in [32]$, and two double blocks $(7, 23) \in [36] \times [28]$ and $(38, 0) \in [40] \times [24]$, and we are ready for Pass 2.

Output of Pass 2. The first singleton $12 \in [32]$ we output immediately, as it fits into 5 bits already. Next, we transform the first double blocks $(7, 23) \in [36] \times [28]$ into a 10-bit number in range $[32^2] = [1024]$ by setting this number to $23 \cdot 36 + 7 = 835$. We can write it as 10 bits, but for better understanding, let's convert it into two numbers in $[B] = [32]$ by decomposing $835 = 26 \cdot 32 + 3$, so we get a double block $(3, 26) \in [32] \times [32]$. We immediately output these numbers.

Similarly, we transform the second (and last) double blocks $(38, 0) \in [40] \times [24]$ into a 10-bit number in range $[32^2] = [1024]$ by setting this number to $0 \cdot 40 + 38 = 38$.[2] We can write it as 10 bits (in fact, 6 bits, see the Footnote 2), but for better understanding, let's convert it into two numbers in $[B] = [32]$. We do it by decomposing $38 = 1 \cdot 32 + 6$, getting a double block $(6, 1) \in [32] \times [32]$, which we immediately output.[3]

Final Output. To summarize, we transform three input blocks $5, 7, 31$ into 5 output blocks $12, 3, 26, 6, 1$.

1.1.2 Decoding

We now show how to decode five blocks $12, 3, 26, 6, 1$ back to $5, 7, 31$.

Recovering Output of Pass 2. We first need to recover the output of Pass 2. The first block 12 is already in Pass 2. We now take the next double block $(3, 26) \in [32] \times [32]$, and convert it into double block in the range $[36] \times [28]$. First, we write our block as one big number $26 \cdot 32 + 3 = 835$. We then decompose $835 = 23 \cdot 36 + 7$, which indeed gives us the correct block $(7, 23) \in [36] \times [28]$.

[2]Since we always append 0 as the last value in the range $[B - 4i - 4]$, the output is always the previous block whose actual range $[B + 4i + 4]$ is only slightly larger than $[B]$. In particular, we need only $b + 1$, and not $2b$, bits for this number. See Section 2.

[3]Once again, notice that the last block is always either 0 or 1, depending if the converted number was less than B or between B and $B + 4i + 3$.

Similarly, we transform the second (and last) double blocks $(6,1) \in [32] \times [32]$ into a block in range $[40] \times [24]$ by first recovering the "large" number $1 \cdot 32 + 6 = 38$, and then decomposing it as $38 = 0 \cdot 40 + 38$. This gives us the next double block $(38, 0)$. Notice, we actually *do not know yet this is the last block*, but let us move to the decoding of Pass 1, which happens in parallel, and which will let us figure this out!

RECOVERING OUTPUT OF PASS 1. As we just saw, we recovered the output $12, 7, 23, 38, 0$ of Pass 1, although we do not know that this is the end yet. How do we find out? Because we actually try to recover the input to Pass 1, as we recover the output of Pass 2 (which is also the output of Pass 1 re-grouped). Let's do it!

After regrouping, we get double blocks $(12, 7) \in [32] \times [36]$, $(23, 38) \in [28] \times [40]$, and not yet (and never!) completed element 0 (which is the last 0, but we do not know it yet). We transform the first block $(12, 7)$ into a large number $7 \cdot 32 + 12 = 236$, and then write $236 = 7 \cdot 33 + 5 = 236$. This gives us the first input block $(5, 7)$, which we immediately output, since none of the numbers is equal to $32 = \text{EOF}$, so we know this is not the end of the file.

We then do the same thing for the second block $(23, 38) \in [28] \times [40]$, transforming it into a large number $38 \cdot 28 + 23 = 1087$, which we then write as $1087 = 32 \cdot 33 + 31$. This gives us the next input block $(31, 32)$. So we output 31 and *stop*, since we just recovered the next symbol 32, which is end-of-file.

Thus, we indeed recovered three blocks $5, 7, 31$, and do know there is no more data coming.

1.2 Additional Properties

Based on Property 1, one can support random access to the encoding in constant time. Decoding a block in the middle of the file requires reading 4 output blocks. Modifying an input block will read and change 4 consecutive output blocks. For instance, to modify block 3 in Figure 1, we first read output blocks 2, ..., 5. From these, we compute the input block 4, and blocks 2 and 5 output by Pass 2. From the new value of block 3 and the old block 4, we can rerun Pass 1 to update the intermediate blocks 3 and 4. We now know the intermediate blocks 2,..., 5, so the output blocks can be computed by running Pass 2.

Appending to the file and truncating can be reduced to write operations.

1.3 Practical Considerations

To ensure fast arithmetic operations, one would set $b = 32$ or $b = 64$. While the algorithm uses arithmetic on double precision ($2b$ bits), it is standard to implement division and multiplication by $2^b \pm x$ using fast, single-precision operations.

For a given b, our basic algorithm can process a stream of up to $n_0 = 2^{(b-1)/2}$ blocks. For $n > n_0$ blocks, we can trivially obtain an encoding with overhead $\frac{n}{n_0} + O(1)$ blocks, by applying SOLE on each chunk of n_0 blocks. Specifically, one block is wasted for each chunk except the last one, where 3 blocks may be wasted.

With a minimal setting of $b = 32$ bits, our encoding adds a block (4 bytes) per each $2^{15.5} = 46,340$ blocks ($\approx$ 181Kb), making our overhead roughly $2^{-15.5} \approx 0.002\%$. For comparison, the naïve encoding will waste 1 bit per block, which is $\frac{1}{32} = 3.125\%$, a factor of 1448 worse than our encoding. E.g., a 32Gb file will have a negligible 707Kb overhead with our encoding, and 1Gb overhead with the naïve encoding.

2 A Tighter Encoding

Note: more advanced material for the "Honors" students.

The previous algorithm wasted up to 3 blocks (when n is even). If $n \geq 2$ and $b \geq 2\log_2 n + 2$, we can instead obtain an optimal encoding that always uses $n+1$ output blocks. The first idea is to conceptually insert the EOF symbol two blocks before the actual termination of the input stream. This can be done by buffering the last two blocks, and ensures that EOF will be followed by two output blocks. Hence, by Property 1, the usual decoding algorithm will certainly observe the EOF without reading past the end of the encoding. At this point, we switch to a special termination procedure for the last two blocks.

Imagine that, after EOF appearing on position $n-1$, and the last two blocks on positions n and $n+1$, an infinite stream of zeros follows in the input stream. Then, it is not hard to see that block $n+2$ of the usual encoding will be 0 or 1 (henceforth "the final bit"), and the remaining blocks are guaranteed to be zero. In the new algorithm, we will output the usual output values of blocks up to $n+1$.

Instead of wasting block $n+2$ for the final bit, we will use the following hack: we will have two end-of-file symbols, EOF_0 and EOF_1, coding this bit. We note that there is no circular dependence, since block $n+2$ (the final bit) does not depend on block $n-1$ (the EOF) by Property 1. The price we pay is increasing the alphabet to $B+2$, instead of $B+1$. Now ranges of the form $[B \pm 4i]$ become $[B \pm 8i]$. The encoding is possible as long as:

$$(B+2)^2 \leq (B-8i)(B+8i+8) \iff B \geq 4(2i+1)^2 - 3$$

Since $2i+1 \leq n$, this requires $b \geq 2\log_2 n + 2$.