

# Molecular Arithmetic Coding (MAC) Pseudo-Codes

Melih Şahin\*

\*Computer Engineering & Mathematics Student at Koç Univeristy. {melihsahin21}@ku.edu.tr  
This work accompanies the research done by Melih Şahin, Beyza E. Ortlek, and Ozgur B. Akan.  
Pdf of our paper can be found at: <https://arxiv.org/abs/2403.04672>

---

**Algorithm 1** Zero-Order Arithmetic Coding for MC With EOF

---

**Require:** the bit-precision  $P$ , word to be encoded  $word$ , a function which maps each number  $k$  to the probability of the  $k$ th symbol,  $prob$

▷ EOF symbol is included in  $prob$

```
1:  $whole = 2^P$ 
2:  $N = prob.size$  ▷  $N$  is the number of symbols
3: for  $i=1,...,N$   $c[i] = \sum_{k=1}^{i-1} prob[k]$ 
4: for  $i=1,...,N$   $d[i] = \sum_{k=1}^i prob[k]$ 
   ▷ determination of intervals for each symbol
5:  $up\_is\_ONE = True$ 
6:  $down\_is\_ZERO = True$ 
7:  $s\_up = 1$ 
8:  $s\_down = 0$ 
9:  $s = 0$ 
10:  $emit = []$  ▷  $emit$  is initially an empty array
11: for  $i = 1$  to  $word.length$  do
12:    $the\_character = word[i]$ 
13:   ( $emit\_new, s, a, b, down\_is\_ZERO, up\_is\_ONE,$ 
      $s\_down, s\_up, increase\_in\_s, pure\_increase$ ) =
     One_Time_Encoder( $whole, c, d, prob, s, a, b,$ 
      $the\_character, down\_is\_ZERO, up\_is\_ONE, s\_up,$ 
      $s\_down$ )
14:    $emit = emit + emit\_new$  ▷  $+$  is the concatenation
     operator. Variables  $increase\_in\_s, pure\_increase$  are
     redundant here and will be used in the decoding algorithm
     later.
15: end for
16:  $emit = emit + \mathbf{One\_Time\_Last\_Character\_Encoder}$ 
   _With_EOF( $whole, s, a, b, down\_is\_ZERO,$ 
    $up\_is\_ONE, s\_up, s\_down$ )
17: return  $emit$ 
```

---

**Algorithm 2** One Time Encoder

---

**Require:**  $whole, c, d, prob, s, a, b, the\_character,$   
 $down\_is\_ZERO, up\_is\_ONE, s\_up, s\_down$

```
1:  $half = round(whole/2)$ 
2:  $inverse\_golden\_ratio = 0.6180339887498948482$ 
3:  $zero\_length = round(inverse\_golden\_ratio * whole)$ 
4:  $one\_length = whole - zero\_length$ 
5:  $Emit = []$ 
6:  $increase\_in\_s = 0$ 
7:  $pure\_increase = 0$ 
8:  $character\_number = prob.index\_of(the\_character)$ 
```

```
9:  $w = b - a$ 
10:  $copy\_a = a$ 
11:  $a = a + round((c[character\_number] * w) / whole)$ 
12:  $b = copy\_a + round((d[character\_number] * w) / whole)$ 
13: if (not  $down\_is\_ZERO$ ) and (not  $up\_is\_ONE$ ) then
14:   if  $b < one\_length$  or  $a \geq one\_length$  then
15:     if  $b < one\_length$  then
16:        $a = round((a * (whole)) / one\_length)$ 
17:        $b = round((b * (whole)) / one\_length)$ 
18:     if  $s > 0$  then
19:        $Emit.append(s\_down)$ 
20:        $new\_commute = 1$ 
21:       if  $s\_down == 1$  then
22:          $new\_commute = 0$ 
23:       end if
24:       for  $t = 0$  to  $s - 2$  do
25:         if  $t \% 2 == 0$  then
26:            $Emit.append(new\_commute)$ 
27:         else
28:            $Emit.append(s\_down)$ 
29:         end if
30:       end for
31:     end if
32:      $Emit.append(10)$ 
33:      $pure\_increase++ = 2$ 
34:      $s = 0$ 
35:   else
36:      $a = round(((a - one\_length) * (zero\_length)) / one\_length)$ 
37:      $b = round(((b - one\_length) * (zero\_length)) / one\_length)$ 
38:     if  $s > 0$  then
39:        $Emit.append(s\_up)$ 
40:       for  $t = 0$  to  $s - 2$  do
41:          $Emit.append(0)$ 
42:       end for
43:     end if
44:      $Emit.append(0)$ 
45:      $pure\_increase++ = 1$ 
46:   end if
47:    $down\_is\_ZERO = True$ 
48:    $up\_is\_ONE = True$ 
49: end if
50: else if ( $down\_is\_ZERO$ ) and ( $up\_is\_ONE$ ) then
```

▷ we will not check if  $s > 0$  as it is logically impossible

```

51:  while  $b < \text{zero\_length}$  or  $a \geq \text{zero\_length}$  do
52:      if  $b < \text{zero\_length}$  then
53:           $\text{Emit.append}(0)$ 
54:           $\text{pure\_increase} += 1$ 
55:           $a = \text{round}((a * (\text{whole})) / \text{zero\_length})$ 
56:           $b = \text{round}((b * (\text{whole})) / \text{zero\_length})$ 
57:      else
58:           $\text{Emit.append}(10)$ 
59:           $\text{pure\_increase} += 1$ 
60:           $a = \text{round}(((a - \text{zero\_length}) * (\text{whole})) / \text{one\_length})$ 
61:           $b = \text{round}(((b - \text{zero\_length}) * (\text{whole})) / \text{one\_length})$ 
62:      end if
63:  end while
64:  else if  $(\text{down\_is\_ZERO})$  and  $(\text{not up\_is\_ONE})$  then
65:      if  $b < \text{half}$  or  $a \geq \text{half}$  then
66:          if  $b < \text{half}$  then
67:              if  $s > 0$  then
68:                   $\text{Emit.append}(s\_down)$ 
69:                   $\text{new\_commute} = 1$ 
70:                  if  $s\_down == 1$  then
71:                       $\text{new\_commute} = 0$ 
72:                  end if
73:                  for  $t = 0$  to  $s - 2$  do
74:                      if  $t \% 2 == 0$  then
75:                           $\text{Emit.append}(\text{new\_commute})$ 
76:                      else
77:                           $\text{Emit.append}(s\_down)$ 
78:                      end if
79:                  end for
80:                   $s = 0$ 
81:              end if
82:               $\text{Emit.append}(0)$ 
83:               $\text{pure\_increase} += 1$ 
84:               $a = 2 * a$ 
85:               $b = 2 * b$ 
86:          end if
87:          if  $a \geq \text{half}$  then
88:              if  $s > 0$  then
89:                   $\text{Emit.append}(s\_up)$ 
90:                  for  $t = 0$  to  $s - 2$  do
91:                       $\text{Emit.append}(0)$ 
92:                  end for
93:                   $s = 0$ 
94:              end if
95:               $\text{Emit.append}(0)$ 
96:               $\text{pure\_increase} += 1$ 
97:               $a = 2 * (a - \text{half})$ 
98:               $b = 2 * (b - \text{half})$ 
99:          end if
100:           $\text{down\_is\_ZERO} = \text{True}$ 
101:           $\text{up\_is\_ONE} = \text{True}$ 
102:      end if
103:  end if
104:  while  $\text{True}$  do
105:      if  $(\text{down\_is\_ZERO})$  and  $(\text{up\_is\_ONE})$  then
106:           $\text{special\_number\_0} = \text{round}((\text{whole} * \text{inverse\_golden\_ratio}) * \text{inverse\_golden\_ratio})$ 
107:          if  $a \geq \text{special\_number\_0}$  then
108:               $a = \text{round}(((a - \text{special\_number\_0}) * (\text{whole})) / (\text{whole} - \text{special\_number\_0}))$ 
109:               $b = \text{round}(((b - \text{special\_number\_0}) * (\text{whole})) / (\text{whole} - \text{special\_number\_0}))$ 
110:          ▷ this part [107-121] is explained in the Fig. 25
111:          if  $s == 0$  then
112:               $s\_up = 1$ 
113:               $s\_down = 0$ 
114:          end if
115:           $s += 1$ 
116:           $\text{increase\_in\_s} += 1$ 
117:           $\text{down\_is\_ZERO} = \text{False}$ 
118:           $\text{up\_is\_ONE} = \text{False}$ 
119:      else
120:          break
121:      end if
122:  else if  $(\text{not down\_is\_ZERO})$  and  $(\text{not up\_is\_ONE})$  then
123:       $\text{special\_number\_1} = \text{round}((\text{whole} * 2 * \text{inverse\_golden\_ratio}) * \text{inverse\_golden\_ratio})$ 
124:      if  $b < \text{special\_number\_1}$  then
125:           $a = \text{round}((a * (\text{whole})) / \text{special\_number\_1})$ 
126:           $b = \text{round}((b * (\text{whole})) / \text{special\_number\_1})$ 
127:          ▷ this part is explained in the Fig. 12
128:          if  $s == 0$  then
129:               $s\_up = 0$ 
130:               $s\_down = 1$ 
131:          end if
132:           $s += 1$ 
133:           $\text{increase\_in\_s} += 1$ 
134:           $\text{down\_is\_ZERO} = \text{True}$ 
135:           $\text{up\_is\_ONE} = \text{False}$ 
136:      else
137:          break
138:      end if
139:  else if  $(\text{down\_is\_ZERO})$  and  $(\text{not up\_is\_ONE})$  then
140:       $\text{special\_number\_2} = \text{round}((\text{whole} / 2) * (1 + \text{inverse\_golden\_ratio}))$ 
141:       $\text{special\_number\_3} = \text{round}((\text{whole} / 2) * \text{inverse\_golden\_ratio})$ 
142:      if  $b < \text{special\_number\_2}$  and  $a \geq \text{special\_number\_3}$  then
143:           $a = \text{round}((a - \text{special\_number\_3}) * 2)$ 
144:           $b = \text{round}((b - \text{special\_number\_3}) * 2)$ 
145:          ▷ this part is explained in the Fig. 3
146:          if  $s == 0$  then

```

```

144:         s_up = 0
145:         s_down = 0
146:     end if
147:     s+ = 1
148:     increase_in_s+ = 1
149:     down_is_ZERO = False
150:     up_is_ONE = False
151: else
152:     break
153: end if
154: end if
155: end while
156: return (Emit, s, a, b, down_is_ZERO, up_is_ONE,
         s_down, s_up, increase_in_s, pure_increase)

```

We will now be figuratively explaining interval rescaling section of Algorithm 2.

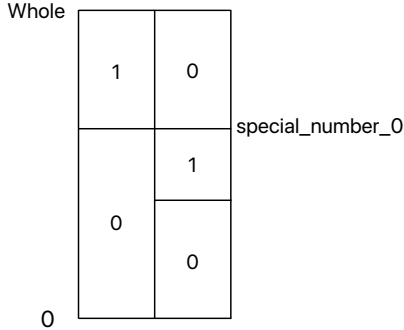


Fig. 1: Case 1

In the case where the current down and up bits are 0 and 1 respectively, if our interval  $[a,b)$  lies above the special\_number\_0 shown in Fig. 1, we are able to rescale our current interval such that the new intervals down bit is 1 and up bit is 0. The details are given in the Algorithm 2[107-121].

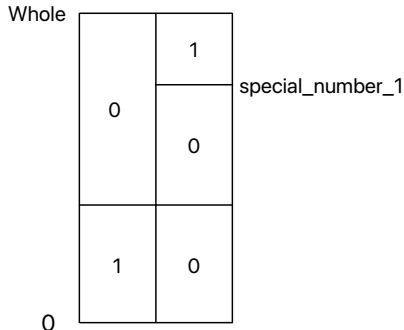


Fig. 2: Case 2

In the case where the current down and up bits are 1 and 0 respectively, if our interval  $[a,b)$  lies below the special\_number\_1 shown in Fig. 2, we are able to rescale our current interval such that the new interval's down and up bits are both 0. The details are given in the Algorithm 2[123-138].

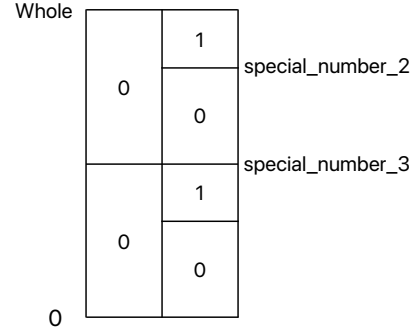


Fig. 3: Case 3

In the case where the current down and up bits are both 0, if our interval  $[a,b)$  lies in between special\_number\_3 and special\_number\_2 shown in Fig. 3, we are able to rescale our current interval such that the new intervals down bit is 1 and up bit is 0. The details are given in the Algorithm 2[139-156].

---

### Algorithm 3 One Time Last Character Encoder With EOF

---

**Require:** *whole, c, d, prob, s, a, b, down\_is\_ZERO, up\_is\_ONE, s\_up, s\_down*

```

1: half = round(whole/2)
2: inverse_golden_ratio = 0.6180339887498948482
3: zero_length = round(inverse_golden_ratio * whole)
4: one_length = whole - zero_length
5: Emit = []
6: while True do
7:     if s > 0 then
8:         if (not down_is_ZERO) and (not up_is_ONE)
           then
9:             if (a == 0 and b >= one_length) or
              (b == whole and a < one_length) then
10:                if b == whole and a < one_length then
11:                    Exact copy of Algorithm_6 [40-45]
12:                break
13:            end if
14:            if a == 0 and b >= one_length then
15:                Exact copy of Algorithm 2 [19-30]
16:            break
17:        end if
18:    else
19:        down_bit_length = one_length - a
20:        upper_bit_length = b - one_length
21:        if down_bit_length >=
          upper_bit_length then
22:            Exact copy of Algorithm 2 [19-30]
23:            Emit.append(10)
24:            b = whole
25:            a = round((a * whole)/one_length)
26:            if a >= zero_length then
27:                Emit.append(10)
28:                break

```

29:	<b>end if</b>	80:	<b>end if</b>
30:	<b>else</b>	81:	<b>else</b>
31:	Exact copy of Algorithm 2 [40-45]	82:	<i>down_bit_length</i> = <i>one_length</i> - <i>a</i>
32:	<i>a</i> = 0	83:	<i>upper_bit_length</i> = <i>b</i> - <i>one_length</i>
33:	<i>b</i> = round((( <i>b</i> - <i>one_length</i> ) * <i>whole</i> ) / (( <i>whole</i> - <i>one_length</i> )))	84:	<b>if</b> <i>down_bit_length</i> >= <i>upper_bit_length</i> <b>then</b>
34:	<b>if</b> <i>b</i> < <i>zero_length</i> <b>then</b>	85:	<i>Emit.append</i> (0)
35:	<i>Emit.append</i> (0)	86:	<i>b</i> = <i>whole</i>
36:	<b>break</b>	87:	<i>a</i> = round(( <i>a</i> * <i>whole</i> ) / <i>one_length</i> )
37:	<b>end if</b>	88:	<b>else</b>
38:	<b>end if</b>	89:	<i>Emit.append</i> (10)
39:	<i>s</i> = 0	90:	<i>a</i> = 0
40:	<i>down_is_ZERO</i> = <i>True</i>	91:	<i>b</i> = round((( <i>b</i> - <i>one_length</i> ) * <i>whole</i> ) / <i>one_length</i> )
41:	<i>up_is_ONE</i> = <i>True</i>	92:	<b>end if</b>
42:	<b>end if</b>	93:	<b>end if</b>
43:	<b>else if</b> (not <i>down_is_ZERO</i> ) and (not <i>up_is_ONE</i> ) <b>then</b>	94:	<b>else if</b> (not <i>down_is_ZERO</i> ) and (not <i>up_is_ONE</i> ) <b>then</b>
44:	<b>if</b> ( <i>a</i> == 0 and <i>b</i> >= <i>half</i> ) or ( <i>b</i> == <i>whole</i> and <i>a</i> < <i>half</i> ) <b>then</b>	95:	<b>if</b> ( <i>a</i> == 0 and <i>b</i> >= <i>one_length</i> ) or ( <i>b</i> == <i>whole</i> and <i>a</i> < <i>one_length</i> ) <b>then</b>
45:	<b>if</b> <i>b</i> == <i>whole</i> and <i>a</i> < <i>half</i> <b>then</b>	96:	<b>if</b> <i>b</i> == <i>whole</i> and <i>a</i> < <i>one_length</i> <b>then</b>
46:	Exact copy of Algorithm 2 [40-45]	97:	<i>Emit.append</i> (0)
47:	<b>break</b>	98:	<b>break</b>
48:	<b>else</b>	99:	<b>else</b>
49:	Exact copy of Algorithm 2 [19-30]	100:	<i>Emit.append</i> (0)
50:	<i>Emit.append</i> (0)	101:	<b>break</b>
51:	<b>break</b>	102:	<b>end if</b>
52:	<b>end if</b>	103:	<b>else</b>
53:	<b>else</b>	104:	<i>down_bit_length</i> = <i>one_length</i> - <i>a</i>
54:	<i>down_bit_length</i> = <i>half</i> - <i>a</i>	105:	<i>upper_bit_length</i> = <i>b</i> - <i>one_length</i>
55:	<i>upper_bit_length</i> = <i>b</i> - <i>half</i>	106:	<b>if</b> <i>down_bit_length</i> >= <i>upper_bit_length</i> <b>then</b>
56:	<b>if</b> <i>down_bit_length</i> >= <i>upper_bit_length</i> <b>then</b>	107:	<i>Emit.append</i> (10)
57:	Exact copy of Algorithm 2 [19-30]	108:	<i>b</i> = <i>whole</i>
58:	<i>Emit.append</i> (0)	109:	<i>a</i> = round(( <i>a</i> * <i>whole</i> ) / <i>zero_length</i> )
59:	<i>a</i> = 2 * <i>a</i>	110:	<b>else</b>
60:	<i>b</i> = <i>whole</i>	111:	<i>Emit.append</i> (0)
61:	<b>else</b>	112:	<i>a</i> = 0
62:	Exact copy of Algorithm 2 [40-45]	113:	<i>b</i> = round((( <i>b</i> - <i>zero_length</i> ) * <i>whole</i> ) / <i>zero_length</i> )
63:	<i>a</i> = 0	114:	<b>end if</b>
64:	<i>b</i> = round(( <i>b</i> - <i>half</i> ) * 2)	115:	<i>down_is_ZERO</i> = <i>True</i>
65:	<b>end if</b>	116:	<i>up_is_ONE</i> = <i>True</i>
66:	<i>s</i> = 0	117:	<b>end if</b>
67:	<i>down_is_ZERO</i> = <i>True</i>	118:	<b>else if</b> ( <i>down_is_ZERO</i> ) and (not <i>up_is_ONE</i> ) <b>then</b>
68:	<i>up_is_ONE</i> = <i>True</i>	119:	<b>if</b> ( <i>a</i> == 0 and <i>b</i> >= <i>half</i> ) or ( <i>b</i> == <i>whole</i> and <i>a</i> < <i>half</i> ) <b>then</b>
69:	<b>end if</b>	120:	<b>if</b> <i>a</i> == 0 and <i>b</i> >= <i>half</i> <b>then</b>
70:	<b>end if</b>	121:	<i>Emit.append</i> (0)
71:	<b>else</b>	122:	<b>break</b>
72:	<b>if</b> ( <i>down_is_ZERO</i> ) and ( <i>up_is_ONE</i> ) <b>then</b>	123:	<b>else</b>
73:	<b>if</b> ( <i>a</i> == 0 and <i>b</i> >= <i>zero_length</i> ) or ( <i>b</i> == <i>whole</i> and <i>a</i> < <i>zero_length</i> ) <b>then</b>	124:	<i>Emit.append</i> (0)
74:	<b>if</b> <i>b</i> == <i>whole</i> and <i>a</i> < <i>zero_length</i> <b>then</b>	125:	<b>break</b>
75:	<i>Emit.append</i> (10)	126:	<b>end if</b>
76:	<b>break</b>		
77:	<b>else</b>		
78:	<i>Emit.append</i> (0)		
79:	<b>break</b>		

```

127:         else
128:             down_bit_length = one_length - a
129:             upper_bit_length = b - one_length
130:             if down_bit_length >=
131:                 upper_bit_length then
132:                     Emit.append(0)
133:                     a = 2 * a
134:                     b = whole
135:             else
136:                 Emit.append(0)
137:                 a = 0
138:                 b = round((b - half) * 2)
139:             end if
140:             down_is_ZERO = True
141:             up_is_ONE = True
142:         end if
143:     end if
144: end while
145: return (Emit)

```

---

**Algorithm 4** Decoder With EOF

---

**Require:** *whole, bit\_sequence, c, d, alphabet, a, b, down\_is\_ZERO (= True), up\_is\_ONE (= True), symbol\_sequence, s\_up (= 1), s\_down (= 0), s (= 0)*

```

1: n = bit_sequence.length()
2: down_inter =  $\sum_{i=1}^n \text{bit\_sequence}[i] * \text{round}(\text{whole} * (1/\phi)^i)$ 
3: if bit_sequence[n] == 0 then
4:     up_inter = down_inter + round(whole * (1/φ)n)
5: else if bit_sequence[n] == 1 then
6:     up_inter = down_inter + round(whole * (1/φ)n-1)
7: end if
8: number = round((down_inter + up_inter)/2)
9: h = symbol ∈ alphabet such that number < d[symbol]
10: if h == NONE then return ERROR end if
11: symbol_sequence.append(h)
12: if h == EOF then return symbol_sequence end if
13: (emit_new, s, a, b, down_is_ZERO, up_is_ONE, s_down, s_up, increase_in_s, pure_increase) =
    One_Time_Encoder(whole, c, d, prob, s, a, b, h, down_is_ZERO, up_is_ONE, s_up, s_down)
14: bit_sequence = bit_sequence[pure_increase + increase_in_s :]
15: return Decoder_With_EOF(whole, bit_sequence, c, d, alphabet, a, b, down_is_ZERO, up_is_ONE, symbol_sequence, s_up, s_down, s)

```

---

**Algorithm 5** Zero-Order Arithmetic Coding for MC Without EOF

---

**Require:** the bit-precision *P*, word to be encoded *word*, a function which maps each number *k* to the probability of the *k*th symbol, *prob*

```

1: Exact copy of Algorithm_5[1-15]
2: emit_last = One_Time_Last_Character_Encoder_Without_EOF(whole, s, a, b, down_is_ZERO,

```

```

    up_is_ONE, s_up, s_down)
3: return emit + emit_last

```

---

**Algorithm 6** Decoder Without EOF

---

**Require:** *whole, bit\_sequence, c, d, alphabet, a, b, down\_is\_ZERO (= True), up\_is\_ONE (= True), symbol\_sequence, s\_up (= 1), s\_down (= 0), s (= 0)*

```

1: Exact copy of Algorithm_8[1-11, 13-14]
2: s_copy = s
3: emit_last = One_Time_Last_Character_Encoder_Without_EOF(whole, s, a, b, down_is_ZERO, up_is_ONE, s_up, s_down)
4: if (bit_sequence == emit_last[s_copy :]) then
5:     return symbol_sequence
6: end if
7: return Decoder_Without_EOF(whole, bit_sequence, c, d, alphabet, a, b, down_is_ZERO, up_is_ONE, symbol_sequence, s_up, s_down, s)

```

---

**Algorithm 7** One Time Last Character Encoder Without EOF

---

**Require:** *whole, c, d, s, a, b, down\_is\_ZERO, up\_is\_ONE, s\_up, s\_down*

```

1: half = round(whole/2)
2: number_of_added_bits = 0
3: inverse_golden_ratio = 0.6180339887498948482
4: zero_length = round(inverse_golden_ratio * whole)
5: one_length = whole - zero_length
6: Emit = []
7: while True do
8:     if (not down_is_ZERO) and (not up_is_ONE) then
9:         if a >= one_length then
10:             if s > 0 then
11:                 Emit.append(s_up)
12:                 for t = 0 to s - 2 do
13:                     Emit.append(0)
14:                 end for
15:                 s = 0
16:             end if
17:             Emit.append(0)
18:             number_of_added_bits += 1
19:             a = round(((a - one_length) * (zero_length))/one_length)
20:             b = round(((b - one_length) * (zero_length))/one_length)
21:             down_is_ZERO = True
22:             up_is_ONE = True
23:         else if b < one_length then
24:             Exact copy of Algorithm 2 [18-33]
25:             number_of_added_bits += 2
26:             a = round((a * (whole))/one_length)
27:             b = round((b * (whole))/one_length)
28:             down_is_ZERO = True
29:             up_is_ONE = True
30:         else
31:             Exact copy of Algorithm 2 [39-45]

```

```

32:     number_of_added_bits += 1
33:     break
34: end if
35: else if (down_is_ZERO) and (not up_is_ONE)
    then
36:     if a >= half then
37:         Exact copy of Algorithm 2 [39-45]
38:         a = 2 * (a - half)
39:         b = 2 * (b - half)
40:         down_is_ZERO = True
41:         up_is_ONE = True
42:     else if b < half then
43:         Exact copy of Algorithm 2 [18-31]
44:         Emit.append(0)
45:         a = 2 * a
46:         b = 2 * b
47:         down_is_ZERO = True
48:         up_is_ONE = True
49:     else
50:         Exact copy of Algorithm 2 [39-45]
51:         break
52:     end if
53: else if (down_is_ZERO) and (up_is_ONE) then
54:     if a >= zero_length then
55:         Emit.append(10)
56:         a = round(((a - zero_length) *
                    whole)/one_length)
57:         b = round(((b - zero_length) *
                    whole)/one_length)
58:     else if b < zero_length then
59:         Emit.append(0)
60:         a = round((a * whole)/zero_length)
61:         b = round((a * whole)/zero_length)
62:     else
63:         Emit.append(10)

```

```

64:         break
65:     end if
66: end if
67: end while
68: return (Emit)

```

---

**Algorithm 8** Arithmetic Coder Suitability Checker With EOF

---

**Require:** the bit-precision  $P$ , word to be encoded  $word$ , a function which maps each number  $k$  to the probability of the  $k$ th symbol,  $prob$

```

1: Encoded_word=Zero_Order_Arithmetic_Coding
  _for_MC_With_EOF( $P, word, prob$ )
  Decoded_word_1=Decoder_Without_EOF( $P,$ 
    Encoded_word,  $prob$ )
2: Excess = [101010...10]  $\triangleright$  the length of Excess is  $P$ 
3: Decoded_word_2=Decoder_Without_EOF( $P,$ 
    Encoded_word + Excess,  $prob$ )
4: if (Decoded_word_1 is equal to  $word$ ) and
   (Decoded_word_2 is equal to  $word$ ) then
5:     return True
6: else
7:     return False
8: end if

```

---

**Algorithm 9** Arithmetic Coder Suitability Checker Without EOF

---

**Require:** the bit-precision  $P$ , word to be encoded  $word$ , a function which maps each number  $k$  to the probability of the  $k$ th symbol,  $prob$

```

1: Encoded_word=Zero_Order_Arithmetic_Coding
  _for_MC_Without_EOF( $P, word, prob$ )
2: Decoded_word=Decoder_Without_EOF( $P,$ 
    Encoded_word,  $prob$ )
3: if Decoded_word is equal to  $word$  then return True
4: else return False end if

```

---