Analysis

For small parameter values, it is straightforward to generate all symbols, to count them, and to put them into a table for decoding. One way to do so is to go through all bit vectors with n bits and store the ones that match the criteria of $BC(n, k, m)$. Decoding can be done by a linear search in the table, but since the table is sorted, a binary search generally works faster.

It is also possible to generate the symbols more directly (without considering 'invalid' bit patterns). Given a symbol's bit pattern it is not too difficult to construct the next one (if it exists).

Counting symbols

A recursive generator, however, is easier to describe. For that purpose, it is convenient to consider the 'run' vector of a symbol that gives the width of each bar in the symbol. The run vector of the symbol in Figure 1 of the task description is 1231. Procedure call $CountBC(n, k)$ adds the number of symbols in $BC(n, k, m)$ to global variable t:

```
var t: longint;

procedure CountBC(p, q: integer);
  var i: integer;
  begin
  if (p > 0) and (q > 0) then
    for i := 1 to m do
      CountBC(p-i, q-1)
  else
    if (p = 0) and (q = 0) then
      t := t + 1
  end { CountBC } ;
...
readln(n, k, m) ; t := 0 ;
CountBC(n, k)
{ t = size of BC(n, k, m) }
```

In the for-loop, the value of i is the width of bar $k - q + 1$ (the leftmost bar when $q = k$). It can be recorded in a global variable to generate the symbols. However, the symbols are not generated in the desired order, since the groups of '0's (light bars) now go from narrow to wide instead of the other way round. This is repaired by replacing the for-loop by

```
if odd(k-q+1) then
  for i := 1 to m do
    CountBC(p-i, q-1)
else
  for i := m downto 1 do
    CountBC(p-i, q-1)
```

The speed of the procedure can be improved by restricting i to a smaller range whenever possible. A lower bound on i is obtained by observing that the maximum number of modules covered by the remaining $q - 1$ bars equals $(q - 1) * m$. Therefore, i must be at least $n - (q - 1) * m$. Considering the minimum width of $q - 1$ bars yields an upper bound, viz. $n - (q - 1)$. That way, also negative values for p are avoided.

Decoding symbols

All symbols in the bar code can first be generated and stored in a table, which is subsequently searched for decoding. However, it seems better to start by sorting the message symbols to be decoded (duplicates complicate this a little bit), and then to generate all symbols and decode the message 'on-the-fly'. The decoded list is finally written in the original order.

Efficiency considerations

All this, however, should be preceded by an analysis of how many symbols there can be in a bar code. We get an upper bound by forgetting about parameter m, or—what comes to the same thing—by taking $m = n$. In that case, it is easy to count the number of symbols, because each symbol is obtained by distributing $k - 1$ dividers over the $n - 1$ boundaries between modules (at most one divider per boundary). Therefore, the size of $BC(n,k,n)$ equals $\binom{n-1}{k-1}$. The maximum size, thus, is $\binom{32}{16}$ which equals 601,080,390. This is obviously not within reach of procedure *CountBC*.

Efficient counting

Observe that *CountBC* makes many recursive calls with the same parameter pairs (p,q). The number of pairs, however, is at most something on the order of $n * k$. Hence, it seems wise to think of *dynamic programming* and to make a table $t[p,q]$ to store the size of $BC(p,q,m)$ with $0 \le p \le n$ and $0 \le q \le k$. The table for $BC(7,4,3)$ is as follows:

p ↓ \ q →	0	1	2	3	4
0	1	0	0	0	0
1	0	1	0	0	0
2	0	1	1	0	0
3	0	1	2	1	0
4	0	0	3	3	1
5	0	0	2	6	4
6	0	0	1	7	10
7	0	0	0	6	16

This table can also be filled iteratively. In fact, the following recurrence relation is not so hard to derive:

$$
\begin{aligned}
t[p,q] &= 1, \text{ if } p = 0 \text{ and } q = 0 \\
t[p,q] &= 0, \text{ if } p \neq 0 \text{ and } q \le 0 \\
t[p,q] &= 0, \text{ if } p \le 0 \text{ and } q \neq 0 \\
t[p,q] &= \sum_{i=1}^{m} t[p-i, q-1], \\
&\quad \text{if } p > 0 \text{ and } q > 0.
\end{aligned}
$$

The sum can be eliminated by observing that

$$
\begin{aligned}
t[p,q] &= t[p-1,q] \\
&+ t[p-1,q-1] \\
&- t[p-m-1,q-1]; \\
&\quad \text{if } p > 0 \text{ and } q > 0:
\end{aligned}
$$

Here is a procedure to do the filling:

```
var
  t: array[0..33, 0..33] of longint;

procedure FillTable;
  var p, q: integer; s: longint;
  begin
  for q := 0 to k do begin
    if q = 0 then t[0, q] := 1
    else { q > 0 } t[0, q] := 0 ;
    for p := 1 to n do begin
      if q = 0 then t[p, q] := 0
      else { p > 0 and q > 0 } begin
        if p < m+1 then s := 0
        else s := t[p-m-1, q-1] ;
        t[p, q] := t[p, q-1] +
                   t[p-1, q-1] - s
        end { else }
      end { for p }
    end { for q }
  end { FillTable } ;
...
readln(n, k, m) ; FillTable
{ t[n, k] = size of BC(n, k, m) }
```

Efficient decoding

Once this table is constructed it can also be used for efficient decoding (without generating all the symbols in a bar code). Observe that the rank of a symbol equals the number of symbols preceding it. Using the table one can count how many symbols precede a given symbol. This is most easily carried out in terms of run vectors. First we present a procedure for reading a bit vector and storing it as a run vector:

```
type
  runvec = array [1..33] of integer;

procedure ReadSymbol(var rv: runvec);
  var b, i, j: integer; c: char;
  begin
  j := 0 ; { bar index }
  for i := 0 to n do begin
    read(c) ;
    b := ord(c) - ord('0') ;
```

```
    { b = 0  or  b = 1 }
    if b <> j mod 2 then begin
      j := j + 1 ;
      rv[j] := 1
      end { then }
    else
      rv[j] := rv[j] + 1
    end { for i } ;
  readln
  end { ReadSymbol } ;
```

As an example we consider how to decode bit vector 1101100, that is run vector 2122, in $BC(7,4,3)$. Here is the list of all *run* vectors in $BC(7,4,3)$:

0: 1312	8: 2212
1: 1320	9: 2220
2: 1213	10: 2113
3: 1222	11: 2122
4: 1231	12: 2131
5: 1123	13: 3211
6: 1132	14: 3112
7: 2311	15: 3121

The symbols preceding 2122 in the list have been divided into three blocks. The first block consists of all (seven) run vectors with a first run that is strictly less than the first run of 2122. In this case there is only one such run, namely 1, and this block consists of all run vectors starting with 1. The block size equals the number of symbols with $7-1=6$ modules and $4-1=3$ bars. Therefore, its size computed as the number of symbols in $BC(6,3,3)$, which we can look up in the precomputed table: $t[6,3]=7$.

The second block consists of all preceding vectors that are equal to 2122 for the first run and whose second run is strictly *bigger* than the second run of 2122 (bigger because light bars are ordered in the opposite direction). In this case there are two such runs, namely 3 and 2. The block consists of all run vectors starting with 23 and 22, and continuing with $4-2=2$ bars over either $7-5=2$ or $7-4=3$ modules. Therefore, its size is computed as the number of symbols in $BC(2,2,3)$ and $BC(3,2,3)$. According to the table this is $t[2,2]+t[3,2]=1+2=3$.

In general, when decoding run vector *rv* with *k* bars, the run vectors preceding it are divided into *k* blocks. Block *q* consists of all run vectors that are equal to *rv* in the first $q-1$ positions and whose *q*-th run is strictly less (for light bars, more) than that of *rv*. The sizes of these blocks are obtained by adding suitable entries from the precomputed table *t*. Note that block *k* is always empty.

Here is a procedure for decoding a run vector:

```
procedure Decode(rv: runvec;
                 var r: longint);
  var p, q, i: integer;
  begin
  p := n ; r := 0 ;
  { p = # remaining modules }
  { r = size of first q blocks }
  for q := 1 to k-1 do begin
    if odd(q) then begin
      for i := 1 to rv[q]-1 do
        if i <= p then
          r := r + t[p-i, k-q]
      end { then }
    else begin
      for i := rv[q]+1 to m do
        if i <= p then
          r := r + t[p-i, k-q]
      end { else } ;
    p := p - rv[q]
    end { for q }
  end { Decode } ;
```

Variations

Produce the list of all symbols in a given bar code (in that case, the parameters must be further restricted, because otherwise the output could be too large). Produce the symbol with largest rank in a given bar code. Given a bar code and a rank, produce the corresponding symbol. Given a bar code and a symbol (not the last one), produce the next symbol. The ordering could be done differently, e.g.

by considering the run vectors. The upper bound on the width of bars could differ for dark and light bars. There can also be a lower bound on the width of bars, possibly depending on the color as well.

As a simple starter we could also ask for the width of a narrowest and a widest bar in any symbol of $BC(n, k, m)$. This can easily be derived without generating all symbols. Note that it is not always 1 and m respectively.

Given two bar codes BC_1 and BC_2 such that the second contains no fewer symbols than the first. Write a program to convert symbols in the first bar code to corresponding symbols (with the same rank) in the second code.

Motivations and Judging

The upper bound of 33 on n, k, and m yields bar-code sizes within the range of 32-bit integers. This also holds for 34, but not for 35. We have choosen 33 instead of 32 in order not to seduce competitors to pack bit vectors in 32-bit variables.

In general, we will test for correct ordering of widths of both dark bars and light bars. However, we will also include a test case where only the symbols with lowest and highest rank are to be decoded. This allows programs that do not get the order right to score points as well.

We will include a test where the message (list of symbols to be decoded) is reverse sorted. We will also include a bar code in which the minimum bar width does not actually occur; similarly for maximimum bar width.

We will include test input allowing approaches that generate all symbols, either by considering all bit patterns and filtering, or more directly. There will also be test input where these approaches will fail, hopefully one by one, as the sizes are increased. Special cases are those where there is only one symbol, with all bars of minimal or maximal width. Another special case is an empty bar code.

The very best competitors should be able to count and decode up to the maximum values. We also include a test case with large parameters for which the bar code is empty. This can also be relatively easily detected.

Tom Verhoeff
Scientific Committee IOI'95