P06_9 Z4

Teorija informacije

Laboratorijska vježba 2019/2020

Korisničke upute - Koder za Hammingove kodove

Izvršive datoteke su imenovane ***hammingCoder_gcc_linux_amd64***, i ***hammingCoder_clang_windows_386.exe***, ovisno o korištenom prevodilačkom sustavu i ciljnoj platformi za izvršavanje.

Program prima dva argumenta naredbenog retka, *n* i *k*, tim redom (to su parametri Hammingovog koda); nakon pokretanja programa moguć je unos proizvoljno dugačkog niza bitova koji treba biti kodiran u kodne riječi koda zadanog parametrima *n* i *k*. Kontrolni bitovi se u kodnoj riječi nalaze na pozicijama 1, 2, 4, 8, i tako dalje; gdje brojanje pozicija bitova počinje od jedinice. Unošenjem parametra *k* koji ne odgovara parametru *n*, uzrokovan je uranjeni završetak programa, ali uz ispis parametra *k* koji odgovara danom *n*; tako da nije potrebno na papiru ili u glavi računati odgovarajući *k*.

Ulazni niz bitova predstavljen je znakovima ***0*** i ***1*** koji mogu biti odvojeni s po volji mnogo znakova ASCII whitespacea (tab, razmak ili novi redak). Prekid unosa se može postići unosom bilo kojeg slova ili niza slova (na primjer, "**gotovo**") prije novog retka.

Nakon toga program prikazuje generirajuću matricu odabranog koda, te redom kodne riječi (uz izvorne bitove od kojih je svaka kodna riječ sačinjena)..

Primjeri izvršavanja

Korisnički unos je **podebljan**.

Počnimo od najjednostavnijeg Hammingovog koda:

```
$ ./hammingCoder_gcc_linux 3 1
Linear block code [n = 3, k = 1] (n = code word length) (k = number of source bits in each code word)
code rate = R(K) = 0.333333

Enter a message in bits (possibly separated by whitespace) to be Hamming coded using the chosen code parameters:

1 0 11 0 111 0
done

Input source message:
101101110

The generator matrix for the code:

111


To encode the entire source input string into codewords, we divide the input string into parts of k or less bits, where the last part's last bits are padded with zeros. Each
input part is multiplied with the generator to produce the corresponding codeword.

Input    1 bits: 1
Output: 111
Input    1 bits: 0
Output: 000
Input    1 bits: 1
Output: 111
Input    1 bits: 1
Output: 111
Input    1 bits: 0
Output: 000
Input    1 bits: 1
Output: 111
Input    1 bits: 1
Output: 111
Input    1 bits: 1
Output: 111
Input    1 bits: 0
Output: 000
```

Isprobajmo neke pogrešne načine zadavanja parametara, kako bismo vidjeli kako program na to reagira:

```
$ ./hammingCoder_gcc_linux
coder: wrong number of arguments, start the program with two arguments, both natural numbers

$ ./hammingCoder_gcc_linux 25 5
coder: wrong input for second argument (k), try 20

$ ./hammingCoder_gcc_linux 25 20
Linear block code [n = 25, k = 20] (n = code word length) (k = number of source bits in each code word)
code rate = R(K) = 0.8

Enter a message in bits (possibly separated by whitespace) to be Hamming coded using the chosen code parameters:

101101110111101111111111110111111111111111111111101111111111111111111111111111101111111111111111111111111111111111111111111111011111111111111111111111111111111111111111111111111
1110
111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111011111111111111111111111111111111111111000000000000000000000000000000000000000
000
fin

Input source message:
```

```
10110111011110111111111111101111111111111111111111101111111111111111111111111111111111101111111111111111111111111111111111111111110111111111111111111111111111111111111111111111111
11101111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111000000000000000000000000000000000000
0000000
```

The generator matrix for the code:

```
1110000000000000000000000
1001100000000000000000000
0101010000000000000000000
1101001000000000000000000
1000000110000000000000000
0100000101000000000000000
1100000100100000000000000
0001000100010000000000000
1001000100001000000000000
0101000100000100000000000
1101000100000010000000000
1000000000000001100000000
0100000000000001010000000
1100000000000001001000000
0001000000000001000100000
1001000000000001000010000
0101000000000001000001000
1101000000000001000000100
0000000100000001000000010
1000000100000001000000001
```

To encode the entire source input string into codewords, we divide the input string into parts of k or less bits, where the last part's last bits are padded with zeros. Each input part is multiplied with the generator to produce the corresponding codeword.

```
Input   20 bits: 10110111011110111111
Output: 0011011101110110110111111
Input   20 bits: 11111110111111111111
Output: 0110111011101111111111111
Input   20 bits: 11111111111011111111
Output: 1111111111111110011111111
Input   20 bits: 11111111111111111111
Output: 0111111111111111111111111
Input   20 bits: 11101111111111111111
Output: 1010110111111111111111111
Input   20 bits: 11111111111111111111
Output: 0111111111111111111111111
Input   20 bits: 11110111111111111111
Output: 1111111001111111111111111
Input   20 bits: 11111111111111111111
Output: 0111111111111111111111111
Input   20 bits: 11111111111111111111
Output: 0111111111111111111111111
Input   20 bits: 01111111111111111111
Output: 1001111111111111111111111
Input   20 bits: 11111111111111111111
Output: 0111111111111111111111111
Input   20 bits: 11111111111111111111
Output: 0111111111111111111111111
Input   20 bits: 11111111111111111111
Output: 0111111111111111111111111
Input   20 bits: 11111111111111111011
Output: 1010111111111110111111011
Input   20 bits: 11111111111111111111
Output: 0111111111111111111111111
Input   20 bits: 11111111111111111100
Output: 1111111111111111111111100
Input   20 bits: 00000000000000000000
Output: 0000000000000000000000000
Input   20 bits: 00000000000000000000
Output: 0000000000000000000000000
Input    1 bits: 0
Output: 0000000000000000000000000
```

Način implementacije programa pomoću vektora bitova omogućava rad (i to brz) s ogromnim matricama generatorima - vidi na sljedećoj strani:

```
$ ./hammingCoder_gcc_linux 100 93
Linear block code [n = 100, k = 93] (n = code word length) (k = number of source bits in each code word)
code rate = R(K) = 0.93

Enter a message in bits (possibly separated by whitespace) to be Hamming coded using the chosen code parameters:

10110111011110111111011111111111111111011111111111111111111111111111111101111111111111111111111111111111111111111110111111111111111111111111111111111111111111111110
11111111111111111111111111111111111111111111111111111111111111111111111111110111111111111111111111111111111111111111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111101111111111111111111111111111111111111111111111111111111111111111
0000011111111111111111100000000000000000000011111000000000000000000000000000000000000001
kraj

Input source message:
101101110111101111110111111111111111110111111111111111111111111111111111011111111111111111111111111111111111111111101111111111111111111111111111111111111111111111101111111111111111
111111111111111111111111111111111111111111111111111111111111011111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
111111111111111111111111111111111111111111111111111111111111110111111111111111111111111111111111111111111111111111111111111111111110000011111111111111111100000000000000000000001111
1000000000000000000000000000000000000001

The generator matrix for the code:

1110000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
1001100000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0101010000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
1101001000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
1000000110000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0100000101000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
1100000100100000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0001000100010000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
1001000100001000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0101000100000100000000000000000000000000000000000000000000000000000000000000000000000000000000000000
1101000100000010000000000000000000000000000000000000000000000000000000000000000000000000000000000000
1000000000000001100000000000000000000000000000000000000000000000000000000000000000000000000000000000
0100000000000001010000000000000000000000000000000000000000000000000000000000000000000000000000000000
1100000000000001001000000000000000000000000000000000000000000000000000000000000000000000000000000000
0001000000000001000100000000000000000000000000000000000000000000000000000000000000000000000000000000
1001000000000001000010000000000000000000000000000000000000000000000000000000000000000000000000000000
0101000000000001000001000000000000000000000000000000000000000000000000000000000000000000000000000000
1101000000000001000000100000000000000000000000000000000000000000000000000000000000000000000000000000
0000000100000001000000010000000000000000000000000000000000000000000000000000000000000000000000000000
1000000100000001000000001000000000000000000000000000000000000000000000000000000000000000000000000000
0100000100000001000000000100000000000000000000000000000000000000000000000000000000000000000000000000
1100000100000001000000000010000000000000000000000000000000000000000000000000000000000000000000000000
0001000100000001000000000001000000000000000000000000000000000000000000000000000000000000000000000000
1001000100000001000000000000100000000000000000000000000000000000000000000000000000000000000000000000
0101000100000001000000000000010000000000000000000000000000000000000000000000000000000000000000000000
1101000100000001000000000000001000000000000000000000000000000000000000000000000000000000000000000000
1000000000000000000000000000000110000000000000000000000000000000000000000000000000000000000000000000
0100000000000000000000000000000101000000000000000000000000000000000000000000000000000000000000000000
1100000000000000000000000000000100100000000000000000000000000000000000000000000000000000000000000000
0001000000000000000000000000000100010000000000000000000000000000000000000000000000000000000000000000
1001000000000000000000000000000100001000000000000000000000000000000000000000000000000000000000000000
0101000000000000000000000000000100000100000000000000000000000000000000000000000000000000000000000000
1101000000000000000000000000000100000010000000000000000000000000000000000000000000000000000000000000
0000000100000000000000000000000100000001000000000000000000000000000000000000000000000000000000000000
1000000100000000000000000000000100000000100000000000000000000000000000000000000000000000000000000000
0100000100000000000000000000000100000000010000000000000000000000000000000000000000000000000000000000
1100000100000000000000000000000100000000001000000000000000000000000000000000000000000000000000000000
0001000100000000000000000000000100000000000100000000000000000000000000000000000000000000000000000000
1001000100000000000000000000000100000000000010000000000000000000000000000000000000000000000000000000
0101000100000000000000000000000100000000000001000000000000000000000000000000000000000000000000000000
1101000100000000000000000000000100000000000000100000000000000000000000000000000000000000000000000000
0000000000000001000000000000000100000000000000010000000000000000000000000000000000000000000000000000
1000000000000001000000000000000100000000000000001000000000000000000000000000000000000000000000000000
0100000000000001000000000000000100000000000000000100000000000000000000000000000000000000000000000000
1100000000000001000000000000000100000000000000000010000000000000000000000000000000000000000000000000
0001000000000001000000000000000100000000000000000001000000000000000000000000000000000000000000000000
1001000000000001000000000000000100000000000000000000100000000000000000000000000000000000000000000000
0101000000000001000000000000000100000000000000000000010000000000000000000000000000000000000000000000
1101000000000001000000000000000100000000000000000000001000000000000000000000000000000000000000000000
0000000100000001000000000000000100000000000000000000000100000000000000000000000000000000000000000000
1000000100000001000000000000000100000000000000000000000010000000000000000000000000000000000000000000
0100000100000001000000000000000100000000000000000000000001000000000000000000000000000000000000000000
1100000100000001000000000000000100000000000000000000000000100000000000000000000000000000000000000000
0001000100000001000000000000000100000000000000000000000000010000000000000000000000000000000000000000
1001000100000001000000000000000100000000000000000000000000001000000000000000000000000000000000000000
0101000100000001000000000000000100000000000000000000000000000100000000000000000000000000000000000000
1101000100000001000000000000000100000000000000000000000000000010000000000000000000000000000000000000
1000000000000000000000000000000000000000000000000000000000000001100000000000000000000000000000000000
0100000000000000000000000000000000000000000000000000000000000001010000000000000000000000000000000000
1100000000000000000000000000000000000000000000000000000000000001001000000000000000000000000000000000
0001000000000000000000000000000000000000000000000000000000000001000100000000000000000000000000000000
1001000000000000000000000000000000000000000000000000000000000001000010000000000000000000000000000000
0101000000000000000000000000000000000000000000000000000000000001000001000000000000000000000000000000
1101000000000000000000000000000000000000000000000000000000000001000000100000000000000000000000000000
0000000100000000000000000000000000000000000000000000000000000001000000010000000000000000000000000000
1000000100000000000000000000000000000000000000000000000000000001000000001000000000000000000000000000
0100000100000000000000000000000000000000000000000000000000000001000000000100000000000000000000000000
1100000100000000000000000000000000000000000000000000000000000001000000000010000000000000000000000000
0001000100000000000000000000000000000000000000000000000000000001000000000001000000000000000000000000
1001000100000000000000000000000000000000000000000000000000000001000000000000100000000000000000000000
0101000100000000000000000000000000000000000000000000000000000001000000000000010000000000000000000000
1101000100000000000000000000000000000000000000000000000000000001000000000000001000000000000000000000
0000000000000001000000000000000000000000000000000000000000000001000000000000000100000000000000000000
1000000000000001000000000000000000000000000000000000000000000001000000000000000010000000000000000000
0100000000000001000000000000000000000000000000000000000000000001000000000000000001000000000000000000
1100000000000001000000000000000000000000000000000000000000000001000000000000000000100000000000000000
0001000000000001000000000000000000000000000000000000000000000001000000000000000000010000000000000000
1001000000000001000000000000000000000000000000000000000000000001000000000000000000001000000000000000
0101000000000001000000000000000000000000000000000000000000000001000000000000000000000100000000000000
1101000000000001000000000000000000000000000000000000000000000001000000000000000000000010000000000000
0000000100000001000000000000000000000000000000000000000000000001000000000000000000000001000000000000
1000000100000001000000000000000000000000000000000000000000000001000000000000000000000000100000000000
0100000100000001000000000000000000000000000000000000000000000001000000000000000000000000010000000000
1100000100000001000000000000000000000000000000000000000000000001000000000000000000000000001000000000
0001000100000001000000000000000000000000000000000000000000000001000000000000000000000000000100000000
1001000100000001000000000000000000000000000000000000000000000001000000000000000000000000000010000000
0101000100000001000000000000000000000000000000000000000000000001000000000000000000000000000001000000
1101000100000001000000000000000000000000000000000000000000000001000000000000000000000000000000100000
0000000000000000000000000000000100000000000000000000000000000001000000000000000000000000000000010000
1000000000000000000000000000000100000000000000000000000000000001000000000000000000000000000000001000
0100000000000000000000000000000100000000000000000000000000000001000000000000000000000000000000000100
1100000000000000000000000000000100000000000000000000000000000001000000000000000000000000000000000010
0001000000000000000000000000000100000000000000000000000000000001000000000000000000000000000000000001
```

```
To encode the entire source input string into codewords, we divide the input string into parts of k or less bits, where the last part's last bits are padded with zeros. Each
input part is multiplied with the generator to produce the corresponding codeword.

Input   93 bits: 101101110111101111110111111111111111110111111111111111111111111111111101111111111111111111111
Output: 0010011001110111110111111011111111111111111101111111111111111111111111111111111101111111111111111111111
Input   93 bits: 111111111111111111011111111111111111111111111111111111111111111111101111111111111111111111111
Output: 0010111111111110111111101111111011111111111111111111111111111111111111111111111111011111111111111111111111111
Input   93 bits: 111111111111111111111111111111111111111111111111111011111111111111111111111111111111111111111
Output: 1010111011111110111111111111111111111111111111111111111110111110111111111111111111111111111111111111
Input   93 bits: 111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
Output: 1110111111111111111111111111111011111111111111111111111111111110111111111111111111111111111111111111
Input   93 bits: 111111111111111111111111111111111111111111110111111111111111111111111111111111111111111111111
Output: 0010111111111110111111111111111111111111111111111101111111111110111111111111111111111111111111111111
Input   93 bits: 111111111111111111000001111111111111111110000000000000000000001111100000000000000000000000000
Output: 1010111111111110111111100000111111111111111111111110000000000000000100000111110000000000000000000000000
Input   13 bits: 0000000000001
Output: 0100000000000001010000000000000000000000000000000000000000000000000000000000000000000000000000000000
```

Izvorni kod programa:

```
// Copyright 2019 Neven Sajko. All rights reserved.
//
// https://github.com/nsajko/hammingCode
//
// A Hamming code coder.
//
// A generator matrix approach is used as an optimization for large
// messages.
//
// Bit vectors are used to compactly represent strings of bits.
//
// For simplicity, I ignored the possibility of heap allocation failing.

#include <stdint.h>
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define nil 0

// All bitVector fields except len are opaque. len is guaranteed to be
// the first field in the struct and of integer type.

////////////////////////////////////////////////////////////////

typedef unsigned
#ifdef FUZZING_BUILD_MODE_UNSAFE_FOR_PRODUCTION
char
#else
long
#endif
bitVectorSmall;
typedef struct {
	// Length and backing storage capacity, both in bits.
	long len, cap;

	// Backing storage. The element type is chosen for efficiency,
	// it enables easily doing arithmetic on bits with great
	// parallelism.
	bitVectorSmall *arr;
} bitVector;

enum {
	// Configurable capacity of the initial input message in bits.
	// Increase for a slight performance boost.
	initialInputMessageCapacity = 1UL << 4,

	// Just for clarity, probably not configurable.
	bitsInAByte = 8,

	bitsInABitVectorSmall = bitsInAByte * sizeof(bitVectorSmall),
```

```c
};

// The set parameter should be either 0 or 1.
//
// If set is 0, effectively nothing is done.
//
// If set is 1, the i-th bit in a is set.
static inline
void
bitVectorSet(bitVector *a, int set, long i) {
	a->arr[i / bitsInABitVectorSmall] |= (bitVectorSmall)set << (i % bitsInABitVectorSmall);
}

// Returns the capacity in bits of the bitVector.
static inline
long
bitVectorCap(const bitVector *a) {
	return a->cap;
}

// Returns ceiling(n / bitsInABitVectorSmall).
static inline
long
ceilDiv(long n) {
	return (n - 1) / bitsInABitVectorSmall + 1;
}

// Clears all the bits in a's capacity.
static inline
void
bitVectorClear(bitVector *a) {
	memset(a->arr, 0, ceilDiv(a->cap) * sizeof(a->arr[0]));
}

// Sets capacity to cap, allocates enough memory to have that capacity
// in bits, sets len to 0. The bits are all initialized to zero.
static inline
void
bitVectorAlloc(bitVector *a, long cap) {
	a->len = 0;
	cap = ceilDiv(cap);
	a->cap = cap * bitsInABitVectorSmall;
	a->arr = calloc(cap, sizeof(a->arr[0]));
}

static
void
bitVectorRealloc(bitVector *a, long cap) {
	cap = ceilDiv(cap);
	bitVectorSmall *tmp = realloc(a->arr, cap * sizeof(a->arr[0]));
	cap *= bitsInABitVectorSmall;
	if (tmp == nil) {
		free(a->arr);
	} else {
		long d = cap - a->cap;
		if (0 < d) {
			memset(&tmp[a->cap / bitsInABitVectorSmall], 0, d / bitsInAByte);
		}
		a->cap = cap;
	}
	a->arr = tmp;
}

static inline
void
bitVectorFree(bitVector *a) {
	free(a->arr);
}

// Performs bitwise exclusive-or between the op and out bit vectors.
static
```

```
void
bitVectorXOR(bitVector *out, const bitVector *op) {
	long i, l = ceilDiv(out->len);
	for (i = 0; i < l; i++) {
		out->arr[i] ^= op->arr[i];
	}
}

// Counts the number of contiguous bits b in the bit vector, starting at
// index i.
static
long
bitVectorCountContiguous(const bitVector *bV, long i, unsigned long b) {
	// The first loop is an optimization for cases where a
	// bitVectorSmall range consists of either 0UL or ~0UL,
	// depending on b.
	long j;
	bitVectorSmall s = 0;
	if (b != 0) {
		s = ~s;
	}
	for (j = i; j < bV->len && (bV->arr[j / bitsInABitVectorSmall] == s); j += bitsInABitVectorSmall) {}
	if (bV->len < j) {
		j = bV->len;
	}
	for (; j < bV->len &&
			(b == (1 & (bV->arr[j / bitsInABitVectorSmall] >> (j % bitsInABitVectorSmall))))
			; j++) {}
	return j - i;
}

// Moves a contiguous range of bits from in starting at index i to out.
static
void
bitVectorMoveInto(bitVector *out, const bitVector *in, long i) {
	long w = i / bitsInABitVectorSmall, y = 0, ly = ceilDiv(out->len), lw = ceilDiv(in->len);
	i %= bitsInABitVectorSmall;
	if (i != 0) {
		for (; y < ly && w + 1 < lw; w++, y++) {
			out->arr[y] = in->arr[w] >> i;
			out->arr[y] |=
				in->arr[w + 1] << ((bitsInABitVectorSmall - i) % bitsInABitVectorSmall);
		}
		if (y < ly) {
			out->arr[y] = in->arr[w] >> i;
		}
	} else {
		for (; y < ly && w < lw; w++, y++) {
			out->arr[y] = in->arr[w];
		}
	}
	i = out->len % bitsInABitVectorSmall;
	ly--;
	out->arr[ly] = (out->arr[ly] << i) >> i;
}

// Shows the boolean argument as bits '0' or '1' on stdout.
static inline
void
printBool(unsigned long b) {
	if (b) {
		putchar('1');
	} else {
		putchar('0');
	}
}

// Shows the bit vector on stdout.
static
void
bitVectorPrint(const bitVector *bV) {
```

```
	// w is for "words", i is for bits.
	long w, i, l = bV->len / bitsInABitVectorSmall;
	for (w = 0; w < l; w++) {
		for (i = 0; i < bitsInABitVectorSmall; i++) {
			printBool((1UL << i) & bV->arr[w]);
		}
	}
	for (i = 0; i < bV->len % bitsInABitVectorSmall; i++) {
		printBool((1UL << i) & bV->arr[w]);
	}
	printf("\n");
}

static
void
bitVectorFillWithInput(bitVector *a, int (*getInput)(void)) {
	bitVectorAlloc(a, initialInputMessageCapacity);
	for (;;) {
		int c = getInput();
		if (c == '	' || c == ' ' || c == '\n' || c == '\r') {
			continue;
		}
		if (c != '0' && c != '1') {
			break;
		}

		c -= '0';

		// c is now either zero or one. Set or clear the
		// corresponding bit accordingly.
		bitVectorSet(a, c, a->len);

		a->len++;

		long cap = bitVectorCap(a);
		if (cap - 1 < a->len) {
			bitVectorRealloc(a, cap << 1);
		}
	}
}

//////////////////////////////////////////////////////////////

// Is l a power of two? Or, equivalently, does l have a set bit count
// (population count/popcount) of one?
static inline
int
isPowerOfTwo(long l) {
	return (l & (l - 1)) == 0;
}

// See below.
static inline
long
hamm(long i) {
	long m = 0, l;
	for (l = 1; l < i; l++) {
		if (!isPowerOfTwo(l)) {
			m++;
		}
	}
	return m;
}

// Returns the corresponding k Hamming code parameter for a given n.
static inline
long
hammingK(long n) {
	return hamm(n) + 1;
}
```

```
// Makes the generator matrix for the [n, k] Hamming code.
static inline
bitVector *
makeGen(long n, long k) {
	bitVector *r = malloc(sizeof(*r) * k);
	long i, j;
	for (i = 0; i < k; i++) {
		bitVectorAlloc(&r[i], n);
		r[i].len = n;
	}
	long nonPowerOfTwoColumns = 0, pow = 1;
	for (j = 0; j < n; j++) {
		if (j + 1 == pow) {
			// j + 1 is a power of two.
			for (i = pow + 1; i <= n; i++) {
				// We check columns that have the pow
				// bit set.
				if (i & pow) {
					bitVectorSet(&r[hamm(i)], 1, j);
				}
			}
			pow <<= 1;
		} else {
			// j + 1 is not a power of two. Set the bit
			// r[row=nonPowerOfTwoColumns, column=j].
			bitVectorSet(&r[nonPowerOfTwoColumns], 1, j);
			nonPowerOfTwoColumns++;
		}
	}
	return r;
}

// Multiplies the row-vector with the matrix.
static inline
void
rowMulMat(bitVector *out, const bitVector *row, const bitVector *mat) {
	bitVectorClear(out);

	// We operate by finding ranges of set bits in the row, prefixed
	// by ranges of unset bits, and then adding up with XOR the
	// corresponding rows from mat.
	long i;
	for (i = 0; i < row->len;) {
		// Skip range of unset bits.
		i += bitVectorCountContiguous(row, i, 0);

		// Find the range of set bits.
		long j = i + bitVectorCountContiguous(row, i, 1);

		// Add up the corresponding range of rows from mat into
		// out.
		long k;
		for (k = i; k < j; k++) {
			bitVectorXOR(out, &(mat[k]));
		}

		i = j;
	}
}

// Frees all k rows belonging to the matrix, then the matrix itself.
static inline
void
freeMat(bitVector *mat, long k) {
	long i;
	for (i = 0; i < k; i++) {
		bitVectorFree(&mat[i]);
	}
	free(mat);
}
```

```c
#ifdef FUZZING_BUILD_MODE_UNSAFE_FOR_PRODUCTION

// This is the fuzzing code, used for finding defects in the main code.
// See https://llvm.org/docs/LibFuzzer.html

static struct {
	long cap;
	const uint8_t *arr;
} fakeGetStorage;

static
int
fakeGet(void) {
	if (fakeGetStorage.cap == 0) {
		return EOF;
	}
	fakeGetStorage.cap--;
	fakeGetStorage.arr++;
	return fakeGetStorage.arr[-1];
}

int
LLVMFuzzerTestOneInput(const uint8_t *data, size_t size) {
	if (size < 3 * sizeof(uint8_t)) {
		return 0;
	}

	uint8_t nByte, kByte;
	memcpy(&nByte, data, sizeof(nByte));
	data += sizeof(nByte);
	size -= sizeof(nByte);
	memcpy(&kByte, data, sizeof(kByte));
	data += sizeof(kByte);
	size -= sizeof(kByte);

	long n = nByte, k = kByte;

	if (k != hammingK(n)) {
		return 0;
	}

	// Initialize fakeGet.
	fakeGetStorage.cap = size;
	fakeGetStorage.arr = data;

	bitVector inMsg;
	bitVectorFillWithInput(&inMsg, fakeGet);

	bitVector *genMat = makeGen(n, k);

	bitVector codeWord;
	bitVectorAlloc(&codeWord, n);
	codeWord.len = n;
	long i;
	bitVector block;
	bitVectorAlloc(&block, k);
	block.len = k;
	for (i = 0; i < inMsg.len; i += k) {
		if (inMsg.len - i < block.len) {
			block.len = inMsg.len - i;
		}

		bitVectorMoveInto(&block, &inMsg, i);

		rowMulMat(&codeWord, &block, genMat);

		bitVectorClear(&block);
	}

	bitVectorFree(&block);
	bitVectorFree(&codeWord);
```

```
	freeMat(genMat, k);
	bitVectorFree(&inMsg);

	return 0;
}

#else

static
int
fgetcStdin(void) {
	return fgetc(stdin);
}

// Shows the array of bit vectors/rows on stdout (as a matrix).
static inline
void
printMatrix(const bitVector *m, long rows) {
	long r;
	for (r = 0; r < rows; r++) {
		bitVectorPrint(&(m[r]));
	}
	printf("\n");
}

// Converts an ASCII char to the number it represents.
static inline
long
ASCIIToNum(long c) {
	return c - '0';
}

// Lexes an ASCII string into a number. Does not look at anything after
// the first char outside the ASCII numeral range.
static inline
long
lexDecimalASCII(const char *s) {
	int i = 0;
	long r = 0;
	for (;; i++) {
		long c = s[i];
		if (c < '0' || '9' < c) {
			break;
		}
		r = 10*r + ASCIIToNum(c);
	}
	return r;
}

int
main(int argc, char *argv[]) {
	// Handle program arguments (argv).
	if (argc != 1 + 2) {
		fprintf(stderr, "coder: wrong number of arguments, start the program with two arguments, "
		                "both natural numbers\n");
		return 1;
	}
	long n = lexDecimalASCII(argv[1]);
	if (isPowerOfTwo(n)) {
		fprintf(stderr, "coder: wrong input for first argument (n). n can not be zero, because no "
		                "code words would exist in that case; and also it can not be a power of "
		                "two, because a parity bit would be wasted in that case as the last bit\n");
		return 1;
	}
	long k = lexDecimalASCII(argv[2]), correctK = hammingK(n);
	if (k != correctK) {
		fprintf(stderr, "coder: wrong input for second argument (k), try %ld\n", correctK);
		return 1;
	}
	fprintf(stderr, "Linear block code [n = %ld, k = %ld] (n = code word length) (k = number of source "
	        "bits in each code word)\ncode rate = R(K) = %g\n", n, k, (double)k / (double)n);
```

```
	// Stdin input of source input message.
	fprintf(stderr, "\nEnter a message in bits (possibly separated by whitespace) to be Hamming coded "
	                "using the chosen code parameters:\n\n");
	fflush(stderr);
	bitVector inMsg;
	bitVectorFillWithInput(&inMsg, fgetcStdin);
	fprintf(stderr, "\nInput source message:\n");
	fflush(stderr);
	bitVectorPrint(&inMsg);
	fflush(stdout);

	// Make and show the code's generator matrix.
	bitVector *genMat = makeGen(n, k);
	fprintf(stderr, "\nThe generator matrix for the code:\n\n");
	fflush(stderr);
	printMatrix(genMat, k);
	printf("\n");
	fflush(stdout);

	fprintf(stderr, "To encode the entire source input string into codewords, we divide the input "
	                "string into parts of k or less bits, where the last part's last bits are padded "
	                "with zeros. Each input part is multiplied with the generator to produce the "
	                "corresponding codeword.\n\n");
	fflush(stderr);
	bitVector codeWord;
	bitVectorAlloc(&codeWord, n);
	codeWord.len = n;
	long i;
	bitVector block;
	bitVectorAlloc(&block, k);
	block.len = k;
	for (i = 0; i < inMsg.len; i += k) {
		if (inMsg.len - i < block.len) {
			block.len = inMsg.len - i;
		}

		// Copy k bits from inMsg to block.
		bitVectorMoveInto(&block, &inMsg, i);
		fprintf(stderr, "Input %4ld bits: ", block.len);
		fflush(stderr);
		bitVectorPrint(&block);
		fflush(stdout);

		// Compute the output code word.
		rowMulMat(&codeWord, &block, genMat);
		fprintf(stderr, "Output: ");
		fflush(stderr);
		bitVectorPrint(&codeWord);
		fflush(stdout);

		// Clear the bit vectors for the next code word.
		bitVectorClear(&block);
	}

	// Deallocate memory.
	bitVectorFree(&block);
	bitVectorFree(&codeWord);
	freeMat(genMat, k);
	bitVectorFree(&inMsg);

	// C main function must return an int, and it should be zero in
	// case no error occured.
	return 0;
}

#endif
```