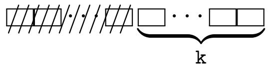
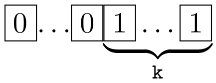
All values are stored in memory using one or several bytes. Bitwise operators allow us to manipulate the bit patterns in integer values of any size (`char`, `int`, `unsigned`, `long`, `int16_t`, `uint32_t`, etc.) and use them to encode/decode values in arbitrary ways.

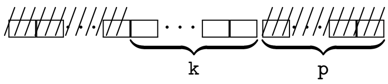
An often-used operation is to obtain the value given by the `k` least significant bits of a number. This means we have to ignore all other bits: 

Bitwise operators work on *all* bits of an integer; they cannot create a bit pattern with fewer bits. To cancel the effect of the bits we don't want, we perform a bitwise AND with a bit pattern that has 1 on the positions we are interested in, and 0 otherwise: 

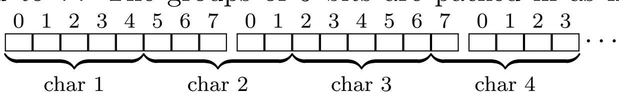
The value of the integer that has this bit pattern is $2^{k-1} + \ldots + 2^1 + 2^0 = 2^k - 1$. If `k` is known, we can simply write this value as an integer constant (it is more readable to do this in hexadecimal or octal). For example, $2^5 - 1$ is 31, or `0x1F`, or `037`.

If `k` is not a compile-time constant, we can rewrite $2^k - 1$ using bit operators: `(1 << k) - 1`, or as `~(~0 << k)`. To understand the last pattern: `~0` has all bits 1; shifted left `k` bits it will have the lower `k` bits 0; complemented, we get the lower bits 1, and the rest 0.

Finally, to obtain the integer represented by the last `k` bits of `n`, we perform a bitwise AND with the number constructed above: `n & ~(~0 << k)`.

Sometimes, we do not need the lower-order `k` bits, but those starting at some bit position `p` (numbered from 0, like the exponents of 2 that the bits correspond to): 

Here, we shift the number right `p` positions before extracting the lower `k` bits: `n >> p & ~(~0 << k)`.

Exercise: Someone wants to represent text using only 5 bits for each character: the values 0 through 31 will encode, in order: `\0`, space (for any whitespace), comma, dash, dot, `?`, the 26 letters (all converted to lowercase). All other characters are mapped to `?`. The groups of 5 bits are packed in as many bytes as needed, using less significant bits first: 

Write a function that takes a string of text and produces a dynamically allocated string with its 5-bit encoding, and another function that decodes and prints a 5-bit-encoded text.

Solution: First, we write two functions that encode and decode a character to/from a 5-bit value:

```
int charenc(int c)
{
  switch (c) {
  case '\0': return 0;
  case ',': return 2;
  case '-': return 3;
  case '.': return 4;
  default: return isspace(c) ? 1 : isalpha(c) ? tolower(c) - 'a' + 6: 5;
  }
}
```

The function directly expresses the encoding rule: it treats the special characters, then the whitespace; the 26 letters starting with `'a'` get values 6 to 31; everything else (including `'?'`) is encoded as 5. The function returns an `int` in the range from 0 to 31 (thus can be represented using 5 bits).

For decoding, we use an array with the characters represented by the values from 0 to 5; the other values up to 31 represent lowercase letters, with 6 used for `'a'`, etc.

```
int chardec(int e)
{
  char dec[6] = "\0 ,-.?";
  return e < 6 ? dec[e] : e - 6 + 'a';
}
```

To encode a string, we encode each character and add the 5 bits to the bits previously obtained; once we have gathered 8 bits we store them and advance in the destination string. Thus we need to remember the bits already encoded and their number.

Assume we have already processed 6 source characters. They yield 6 * 5 = 30 bits of encoding. Of these, 24 bits have already been stored as 3 bytes in the destination string, with 6 bits remaining. Assume these 6 remaining bits are |0|1|0|1|1|0|, from high to low, and the next character to be encoded is `'e'`. It is encoded as `'e' - 'a' + 6 = 4 + 6 = 10`, that is, |0|1|0|1|0|. We shift it left by 6 bits and append (OR) it to the previous 6 bits, the result being |0|1|0|1|0|0|1|0|1|1|0|. We take the low-order 8 bits |1|0|0|1|0|1|1|0| and store them as a byte in the destination string. We are left with the top 3 bits, |0|1|0|, and the processing continues.

This is coded in the function below. We need memory for the encodings (5 bits) of all characters in the string, including the terminator `'\0'`. For `b` bits we need $\lceil b/8 \rceil$ bytes, i.e., `(b + 7)/8`. We need two variables for the encoded bitpattern not yet stored, and the number of bits in it. Every character adds 5 bits; once 8 bits are reached, one byte is stored and those bits are discarded. Processing is done up to and including the null terminator byte of the original string.

```
char *encstr(const char *s)
{
  char *d = malloc((5 * (strlen(s) + 1) + 7) >> 3);
  if (!d) return NULL;
  int bitcnt = 0, bitpart = 0, idx = 0; // bits and part of char already built
  do {
    bitpart |= charenc(*s) << bitcnt; // put next 5 bits in correct position
    if ((bitcnt += 5) > 7) {          // have filled one byte
      d[idx++] = bitpart;     // store the byte
      bitpart >>= 8;          // get rid of bits stored
      bitcnt -= 8;
    }
  } while (*s++);             // exit after encoding \0
  if (bitcnt) d[idx++] = bitpart;   // store remaining part
  return d;
}
```

Decoding proceeds similarly; this time, we get 8 bits (a byte) at a time from the string, and process groups of 5 bits. Again, we have a variable for the bitpattern already extracted and yet to be processed, and another variable for the bit count. Whenever we have less than 5 bits, we get one more byte and place its bits in higher-order position relative to the existing bits. The cast to `unsigned` is needed, otherwise a negative signed char when expanded to an int will be filled with extra bits of 1. Each iteration consumes 5 bits and shifts the bit pattern 5 positions to the right.

```
void decstr(const char *s)
{
  for (int e, bitcnt = 0, bitpart = 0;; bitpart >>= 5, bitcnt -= 5) {
    if (bitcnt < 5) {                 // get more bits
      bitpart |= (unsigned)*s++ << bitcnt;  // add to existing ones
      bitcnt += 8;
    }
    if ((e = bitpart & 0x1F)) putchar(chardec(e)); else break;
  }
  putchar('\n');
}
```

Finally, we combine the functions and check that encoding and then decoding yields the original string.

```
int main(void)
{
  char *enc = encstr("ana are mere multe");
  decstr(enc);
  free(enc);
  return 0;
}
```