05OGDLP - Algorithms & Programming
Laboratory 2

Learning objectives

- C main types and formatted I/O
- Arrays, multidimensional arrays, strings
- Looping and branching

1 Binary to decimal conversion

Convert a binary number, represented as a string (e.g. `101010`), to its decimal equivalent using first principles.

Given a binary input string, your program should produce a decimal output. The program should handle invalid inputs.

Decimal is a base-10 system. A number, e.g. 23, in base 10 notation can be understood as a linear combination of powers of 10:

- The rightmost digit gets multiplied by
$$10^0 = 1$$
- The next number gets multiplied by
$$10^1 = 10$$
- ...
- The nth number gets multiplied by
$$10^{(n-1)}$$
.
- All these values are summed.

So:
$$23 => 2 * 10^1 + 3 * 10^0 => 2 * 10 + 3 * 1 = 23 base 10$$

Binary is similar, but uses powers of 2 rather than powers of 10:

$$101 => 1 * 2^2 + 0 * 2^1 + 1 * 2^0 => 1 * 4 + 0 * 2 + 1 * 1 => 4 + 1 => 5 base 10$$

.

Note: implement the conversion yourself. Do not use something else to perform the conversion for you.

2 Luhn

Given a number determine whether or not it is valid per the Luhn formula.

The Luhn algorithm is a simple checksum formula used to validate a variety of identification numbers, such as credit card numbers and Canadian Social Insurance Numbers.

The task is to check if a given string is valid. Strings of length 1 or less are not valid. Spaces are allowed, but they should be removed before checking. All other non-digit characters are disallowed.

Example 1: valid credit card number

```
4539 3195 0343 6467
```

The first step of the Luhn algorithm is to double every second digit, starting from the right. We will be doubling:
`4_3_ 3_9_ 0_4_ 6_6_`

If doubling the number results in a number greater than 9, then subtract 9 from the product. The results of our doubling:
`8569 6195 0383 3437`

Then sum all of the digits:
`8+5+6+9+6+1+9+5+0+3+8+3+3+4+3+7 = 80`

If the sum is evenly divisible by 10, then the number is valid. This number is valid!

Example 2: invalid credit card number

`8273 1232 7352 0569`
Double the second digits, starting from the right:
`7253 2262 5312 0539`

Sum the digits:
`7+2+5+3+2+2+6+2+5+3+1+2+0+5+3+9 = 57`

57 is not evenly divisible by 10, so this number is not valid.

(Source)

3 Simple encoder/decoder

Write a C program to encode and decode strings. It should be able to

- encode a cleartext string read from `stdin` (use `scanf`) so that each group of subsequent equal symbols is represented by the integer amount and the symbol itself (i.e. `aaabbc` would be encoded as `3a2b1c`)
- decode an encoded string back to its cleartext form (e.g., `6x1y2z` would be decoded as `xxxxxxyzz`)

Assume only `[a-zA-Z]` symbols are used in the cleartext, and each symbol is repeated 9 times at most.

Examples

```
(E)ncode (D)ecode e(X)it >>> e
String to encode >>> ATAAAAAATTTAACCGAGT
Encoded string: 1A1T6A3T2A2C1G1A1G1T

(E)ncode (D)ecode e(X)it >>> D
String to decode >>> 1A1T1C1G5C1T1G2C1T5G2C1A1T
Decoded string: ATCGCCCCCTGCCTGGGGGCCAT
```

4 Words counter

Using `fgets`, read a sentence from `stdin`. Print out the number of distinct words (i.e. `word` is the same as `Word` and `WORD`), discarding space and punctuation. For each word, print its absolute frequency, i.e., the number of times it appears in the text.

Example

Input: *one two three, One. TWO—hey, this is fun Fun FUN!*

```
7 distinct words:
one (2)
two (2)
three (1)
hey (1)
this (1)
is (1)
fun (3)
```