E7: Introduction to Computer Programming for Scientists and Engineers
University of California at Berkeley, Spring 2017
Instructor: Lucas A. J. Bastien

Lab Assignment 05: Binary Representation of Data; Cell and Struct Arrays
Version: release

Due date: Friday February 24th 2017 at 12 pm (noon).

General instructions, guidelines, and comments:

- For each question, you will have to write and submit one or more Matlab functions. We provide a number of test cases that you can use to test your function. The fact that your function works for all test cases provided does not guarantee that it will work for all possible test cases relevant to the question. It is your responsibility to test your function thoroughly, to ensure that it will also work in situations not covered by the test cases provided. During the grading process, your function will be evaluated on a number of test cases, some of which are provided here, some of which are not.
- Submit on bCourses one m-file for each function that you have to write. The name of each file must be the name of the corresponding function, with the suffix `.m` appended to it. For example, if the name of the function is `my_function`, the name of the file that you have to submit is `my_function.m`. **Carefully check the name of each file that you submit.** Do not submit any zip file. If you re-submit a file that you have already submitted, bCourses may rename the file by adding a number to the file's name (*e.g.,* rename `my_function.m` into `my_function-01.m`). This behavior is okay and should be handled seamlessly by our grading system. Do not rename the file yourself as a response to this behavior.
- A number of optional Matlab toolboxes can be installed alongside Matlab to give it more functionality. All the functions that you have to write to complete this assignment can, however, be implemented without the use of any optional Matlab toolboxes. We encourage you to not use optional toolboxes to complete this assignment. All functions of the Matlab base installation will be available to our grading system, but functions from optional toolboxes may not. If one of your function uses a function that is not available to our grading system, you will loose all points allocated to the corresponding part of this assignment. To guarantee that you are not using a Matlab function from an optional toolbox that is not available to our grading system, use one or both of the following methods:
  - ◇ Only use functions from the base installation of Matlab.
  - ◇ Make sure that your function works on the computers of the 1109 Etcheverry Hall computer lab. All the functions available on these computers will be available to our grading system.

- For this assignment, the required submissions are:
  - ◇ `my_morse_to_word.m`
  - ◇ `my_binary_detector.m`
  - ◇ `my_single_to_decimal.m`
  - ◇ `my_ddc.m`
  - ◇ `my_sort_bookshelf.m`
  - ◇ `my_cell_extractor.m`

1. Morse code

Morse Code was used in early radio communication before it was possible to transmit voiced messages. In Morse Code, each letter or digit is represented as a succession of dots and/or dashes. Morse Code can therefore be thought of as a binary representation of letters and numbers. International Morse Code represents letters from A to Z and digits from 0 to 9, and the corresponding codes are shown on Figure 1. In this question, we refer to a "Morse Code letter" as the binary representation of a letter using zeros (for dots) and ones (for dashes), while ignoring spaces between symbols, letters, and words.

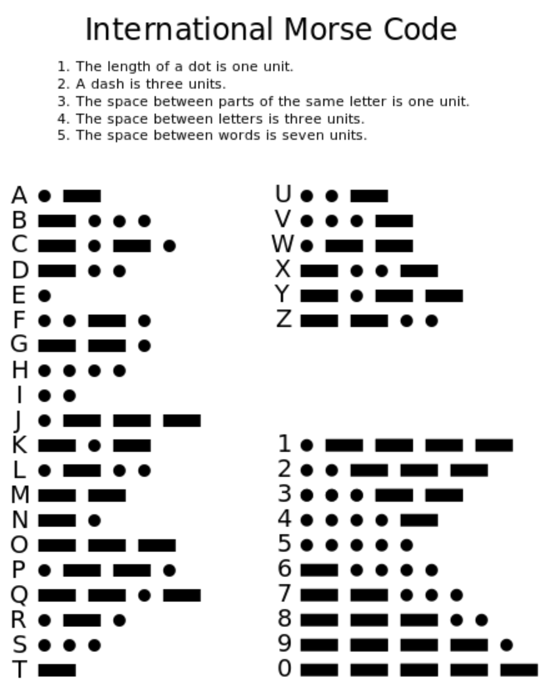

Figure 1: International Morse code. The picture is in the public domain of the United States, and was retrieved from https://en.wikipedia.org/wiki/File:International_Morse_Code.svg on February 13, 2017.

In this question, we only consider letters whose Morse Code representation consists of exactly three symbols (*i.e.* the letters D,G, K, O, R, S, U, and W). We will represent these three-character Morse Code letters in Matlab by 1×3 arrays of class `double` where dots are represented by zeros and dashes are represented by ones. We will represent a sequence of n Morse Code letters in Matlab as a $n \times 3$ array of class `double`, where each row represents a Morse Code letter and the rows have been concatenated vertically to form words. For example, the letter "D" is represented by the 1×3 array: `[1, 0, 0]` and the word "DOG" is represented by the 3×3 array: `[1, 0, 0; 1, 1, 1; 1, 1, 0]`.

Write a function with the following header:

```
function [word] = my_morse_to_word(morse)
```

where:

- `morse` is a $n \times 3$ array of class `double` that is the Morse Code representation of a word as described above. You can assume that $n > 0$.
- `word` is a $1 \times n$ row vector of class `char` that represents the word whose Morse Code representation is described by `morse`. The letters in `word` should all be in upper case.

Test cases:

```
>> my_morse_to_word([1, 0, 0; 1, 1, 1; 1, 1, 0])

ans =

DOG

>> my_morse_to_word([0, 0, 0; 0, 1, 1; 1, 1, 1; 0, 1, 0; 1, 0, 0])

ans =

SWORD

>> my_morse_to_word([0, 1, 1; 1, 1, 1; 0, 1, 0; 1, 0, 1])

ans =

WORK
```

2. Binary representation of integers

In this question, we call an n-bit binary representation an ordered sequence of n zeros and/or ones. Even for a fixed value of n, there are many ways to represent integers using n-bit binary representations. In this question, you will manipulate three different binary representations for integers: "unsigned representation", "sign-magnitude representation", and "two's complement representation". Each of these representations is described below.

In the **unsigned** n-bit representation, each bit represents a power of 2, from 2^0 (right-most bit) to 2^{n-1} (left-most bit). For example, the 8-bit binary representation 00100100 represents the integer $2^2 + 2^5 = 36$, and the 8-bit binary representation 11101000 represents the integer $2^7 + 2^6 + 2^5 + 2^3 = 232$. Note that negative integers cannot be represented with the unsigned n-bit representation.

In the **signed-magnitude** n-bit representation, the left-most bit represents the sign of the integer. If the left-most bit is 0, then the integer is positive. If the left-most bit is 1, then the integer is negative. The other bits represent the magnitude of the integer, each bit representing a power of 2, from 2^0 (right-most bit) to 2^{n-2} (second bit from the left). For example, the 8-bit binary representation 00100100 still represents the number $2^2 + 2^5 = 36$, while the 8-bit binary representation 11101000 now represents the number $-(2^6 + 2^5 + 2^3) = -104$.

In the **two's complement** representation, the left-most bit represents -2^{n-1} and the other seven bits are used in the same way as in the signed-magnitude representation. For example, the 8-bit binary representation 00100100 still represents the number 36, while the 8-bit binary representation 11101000 now represents the number $-2^7 + 2^6 + 2^5 + 2^3 = -24$.

Write a function with the following header:

```
function [representations] = my_binary_detector(binary, integer)
```

where:

- `binary` is $1 \times n$ row vector of class `char` (*i.e.* a character string with n characters) that contains only zeros (character `'0'`) and/or ones (character `'1'`). You can assume that $n > 1$.
- `integer` is a scalar of class `double` that represents an integer (positive, negative, or zero).
- `representations` is a $1 \times m$ `cell` array (with $m \leqslant 3$) that contains 0, 1, 2, or all 3 of the following character strings:
  - ◇ `'unsigned'`. `representations` should contain the character string `'unsigned'` if and only if the binary representation described by the input parameter `binary` is the unsigned n-bit binary representation of the number described by the input parameter `integer`.
  - ◇ `'signmagnitude'`. `representations` should contain the character string `'signmagnitude'` if and only if the binary representation described by the input parameter `binary` is the sign-magnitude n-bit binary representation of the number described by the input parameter `integer`.
  - ◇ `'twoscomplement'`. `representations` should contain the character string `'twoscomplement'` if and only if the binary representation described by the input parameter `binary` is the two's complement n-bit binary representation of the number described by the input parameter `integer`.

If `representations` contains more that one character string, then the character strings should be ordered in alphabetical order. If `representations` does not contain any values, its size should be 0×0, **not** 1×0. You can create a 0×0 `cell` array using the syntax `{}` (*i.e.* curly braces without anything in between).

You may **not** use Matlab's built-in functions `sort`, `bin2dec`, `dec2bin`, `hex2dec`, `dec2hex`, `hex2num`, `num2hex`, `base2dec`, and `dec2base` in this question.

Test cases:

```
>> representations = my_binary_detector('11001011', 203)
representations =
  cell
    'unsigned'

>> representations = my_binary_detector('01001011', 75)
representations =
  1x3 cell array
    'signmagnitude'    'twoscomplement'    'unsigned'

>> representations = my_binary_detector('1111', 7)
representations =
  0x0 empty cell array
```

显示的顺序不一样可以么?

```
>> my_binary_detector('0111', 7)
representations =
  1x3 cell array
    'signmagnitude'    'twoscomplement'    'unsigned'

>> representations = my_binary_detector('0000000000', 0)
representations =
  1x3 cell array
    'signmagnitude'    'twoscomplement'    'unsigned'
```

3. Binary representation of floating point numbers (IEEE-754)

There are multiple ways to represent floating point numbers in binary format (i.e. with only zeros and ones). The Institute of Electrical and Electronics Engineers (IEEE) defined a standard (called IEEE-754) for representing floating point numbers in binary format. IEEE-754 specifies different formats, depending on how many bits (e.g. 16 bits, 32 bits, 64 bits, 128 bits) are used to represent each floating point number. The formats that use 32 bits and 64 bits to represent each number are commonly known as "single precision" and "double precision", respectively. In Matlab, you can experiment by defining a variable `a` that contains the value 1 (`a = 1;`) in the command window and then using the function `whos` to inspect the variables currently defined in the workspace. You should see that the class of variable `a` is `double` (short for double precision) and that it occupies 8 bytes = 64 bits of memory. In this problem, we only condider 32-bits representations, where each number is represented using 32 bits (i.e. a sequence of 32 zeros and/or ones). We index the bits from left to right: the left-most bit is the 1st bit and the right-most bit is the 32nd bit. In the IEEE-

754 standard, the number represented by a sequence of 32 bits can be calculated using the following formulae:

$$\begin{array}{ll} (-1)^s \times 2^{e-d} \times (1+f) & \text{if } e \neq 0 \text{ and } e \neq 2d+1 \\ (-1)^s \times 2^{1-d} \times f & \text{if } e = 0 \text{ and } f \neq 0 \\ 0 & \text{if } e = 0 \text{ and } f = 0 \\ (-1)^s \infty & \text{if } e = 2d+1 \text{ and } f = 0 \\ \text{NaN (Not a Number)} & \text{if } e = 2d+1 \text{ and } f \neq 0 \end{array}$$

where:

- s is the value of the first bit;
- $d = 127$;
- e is given by the integer represented by bits 2 through 9, using the unsigned 8-bit integer representation (see Question 2); and
- The value of f is calculated using bits 10 through 32, each bit representing a negative power of 2, from 2^{-1} (10^{th} bit) to 2^{-23} (32^{nd} bit).

Write a function with the following header:

```
function [result] = my_single_to_decimal(binary)
```

where:

- `binary` is 1×32 row vector of class `char` (*i.e.* a character string made of 32 characters) that can only contain zeros (character `'0'`) and/or ones (character `'1'`).
- `result` is a scalar of class `double` that represents the floating point number (in base 10) that is represented by `binary` using the IEEE-754 32-bit single precision binary representation.

Test cases:

```
>> result = my_single_to_decimal('00111111111110000000000000000000')
result =
    1.9375

>> result = my_single_to_decimal('10111111010000000000000000000000')
result =
   -0.7500

>> result = my_single_to_decimal('00100000100001000000000000000000')
result =
   2.2362e-19
```

```
>> result = my_single_to_decimal('11111111100000000000000000000000')
result =
  -Inf

>> result = my_single_to_decimal('11111111100000000000000110000000')
result =
   NaN
```

4. The Dewey Decimal Classification

The Dewey Decimal Classification (DDC) is a method of sorting books, and more generally areas of knowledge, that is flexible, and continuously updated. It was conceived in the late nineteenth century by Melvil Dewey. In the DDC, an entry such as a book is classified using a number from 0 to 999.9999 . . . (there can be as many decimal digits as necessary), where each digit represents a separate subcategory of the category represented by the digit located to its left. As a consequence, the left-most digit of this DDC number represents the coarser classification criterion, the second digit to the left represents subcategories of the category represented by the left-most digit, and so on.

An advantage of this system is that the number of digits after the decimal place can be adjusted based on how fine of a resolution a library needs to categorize books in a particular field. In this question we focus on the first three digits of the DDC number (*i.e.* the digits to the left of the decimal point). The three digits to the left of the decimal point are used to describe a book's general category, from 000 for computer science, information and general works, to 999 for extraterrestrial worlds. The first (*i.e.* left-most) digit is known as the "first summary", and represents the ten coarser categories of the DDC. The second digit is known as the "second summary", and breaks each of these coarse categories into ten subcategories. The third (*i.e.* right-most) digit is known as the "third summary", and represents further subcategories of the second summary. If two books have the same DDC number, they are sorted alphabetically according to the author's name.

Note: the DDC is copyrighted to the OCLC Online Computer Library Center, Inc. See http://www.oclc.org/en/dewey/features/summaries.html for more information on the DDC.

In this question, you will first write a function that determines the DDC number of books, and then write another function that sorts books according to their DDC numbers.

4.1. Unsorted bookshelf

Write a function with the following header:

```
function [bookshelf] = my_ddc(books)
```

where:

- `books` is a non-empty $n \times 4$ `cell` array where each row represents a book. You can assume that $n > 0$. A book is represented by the 4 following quantities (the 4 elements of the corresponding row in `books`, in this order):

1. A non-empty row vector of class `char` that represents the first summary of the book, among the values:
   - ◇ `'Language'` (the corresponding digit is 4)
   - ◇ `'Literature'` (the corresponding digit is 8)
2. A non-empty row vector of class `char` that represents the second summary of the book, among the values:
   - ◇ `'English'` (the corresponding digit is 2)
   - ◇ `'French'` (the corresponding digit is 4)
   - ◇ `'Italian'` (the corresponding digit is 5)
3. A non-empty row vector of class `char` that represents the third summary of the book. If the first summary of the book is `'Language'`, then the possible values for the third summary are:
   - ◇ `'Phonology'` (the corresponding digit is 1)
   - ◇ `'Etymology'` (the corresponding digit is 2)
   - ◇ `'Grammar'` (the corresponding digit is 5)

   If the first summary of the book is `'Literature'`, then the possible values for the third summary are:
   - ◇ `'Poetry'` (the corresponding digit is 1)
   - ◇ `'Drama'` (the corresponding digit is 2)
   - ◇ `'Speeches'` (the corresponding digit is 5)
4. A non-empty row vector of class `char` that represents the title of the book.

- `bookshelf` is a $2 \times n$ `cell` array where each column represents one of the books described in the input parameter `books`. In each of these columns, the value in the first row should be a scalar of class `double` that represents the DDC number of the book (the first three digits only), and the value in the second row should be a non-empty row vector of class `char` that represents the title of the book. The list of books represented by `bookshelf` should be the same as the list of books represented by `books`, in the same order. Note that the categories and subcategories described above are only a subset of the actual DDC. Here, if a book cannot be classified into the categories and subcategories described above, then its DDC number should be set to `NaN` and its title should be set to the character string `'Discard'`.

Test cases:

```
>> books = {'Language', 'English', 'Etymology', 'Where words come from'};
>> bookshelf = my_ddc(books)
bookshelf =
  2x1 cell array
    [                    422]
    'Where words come from'
```

```
>> books = {'Language', 'English', 'Etymology', 'Where words come from'; ...
            'Literature', 'Italian', 'Poetry', 'Poesie'; ...
            'Literature', 'English', 'Speeches', 'Programming is useful'; ...
            'Literature', 'English', 'Speeches', 'Go E7!'};
>> bookshelf = my_ddc(books)
bookshelf =
  2x4 cell array
    [                  422]    [   851]    [                    825]    [   825]
    'Where words come from'    'Poesie'    'Programming is useful'    'Go E7!'

>> books{1,3} = 'Science-Fiction';
>> bookshelf = my_ddc(books)
bookshelf =
  2x4 cell array
    [    NaN]    [   851]    [                    825]    [   825]
    'Discard'    'Poesie'    'Programming is useful'    'Go E7!'
```

4.2. Sorted bookshelf

Write a function with the following header:

```
function [sorted_bookshelf] = my_sort_bookshelf(bookshelf)
```

where:

- `bookshelf` is a non-empty $2 \times n$ `cell` array (you can assume than $n > 0$) where each column represents one book. In each of these columns, the value in the first row is a scalar of class `double` that represents the DDC number of the book (the first three digits only), and the value in the second row is a non-empty row vector of class `char` that represents the title of the book.
- `sorted_bookshelf` is a $2 \times n$ `cell` array where each column represents one book using a format similar to the format used for the input parameter `bookshelf`. `sorted_bookshelf` should represent the same list of books as `bookshelf`, except that the books should be sorted by increasing DDC number. Books whose DDC number is set to `NaN` should come last. If two or more books share the same DDC number, then they should be ordered in the alphabetical order of their title. This rule also applies when two or more books have their DDC numbers set to `NaN`.

You are allowed to use Matlab's built-in function `sort` for this problem. This built-in function can sort in increasing order row vectors of class `double`. It can also sort in alphabetical order character strings placed in a $1 \times m$ `cell` array.

Note that the actual DDC system specifies that if two or more books share the same DDC number, then they should be ordered in the alphabetical order of their author's last name. In this question, we use the book's title instead of the author's last name to sort books which have the same DDC number.

Test cases:

```
>> bookshelf = {422, 851, 825, 825; 'Where words come from',...
                'Poesie', 'Programming is useful', 'Go E7!'};
>> sorted_bookshelf = my_sort_bookshelf(bookshelf)
sorted_bookshelf =
  2x4 cell array
    [                   422]    [   825]    [                     825]    [   851]
    'Where words come from'    'Go E7!'    'Programming is useful'    'Poesie'

>> bookshelf{1,1} = NaN;
>> bookshelf{2,1} ='Discard';
>> sorted_bookshelf = my_sort_bookshelf(bookshelf)
sorted_bookshelf =
  2x4 cell array
    [   825]    [                     825]    [   851]    [    NaN]
    'Go E7!'    'Programming is useful'    'Poesie'    'Discard'
```

5. Cell array extractor

Write a function with the following header:

```
function [upper_case, lower_case, numbers, special] = my_cell_extractor(array)
```

where:

- `array` is a $1 \times n$ `cell` array. Each element of `array` is either:
  - ◇ A non-empty row vector of class `double`.
  - ◇ A non-empty row vector of class `char` (*i.e.* a non-empty character string).
  - ◇ A non-empty `cell` array that follows the same format as `array`.
- `upper_case` is a non-empty row vector of class `char` that contains all the upper case characters found in the elements of the input parameter `array`, in the order in which they appear in `array`.
- `lower_case` is a non-empty row vector of class `char` that contains all the lower case characters found in the elements of the input parameter `array`, in the order in which they appear in `array`.
- `numbers` is a non-empty row vector of class `double` that contains all the elements of class `double` (except for `NaN`, `-Inf`, and `Inf`) found in the elements of the input parameter `array`, in the order in which they appear in `array`.
- `special` is a non-empty row vector of class `double` that contains all the elements of class `double` among `NaN`, `-Inf`, and `Inf`, that are found in the elements of the input parameter `array`, in the order in which they appear in `array`.

Treat lower case letters from a to z as lower case characters, and any other character as upper case. You can assume that you will find in `array`:

- At least one lower case character; and

- At least one upper case character; and
- At least one element of class `double` that is not `NaN`, `-Inf`, or `Inf`; and
- At least one element of class `double` that is one of `NaN`, `-Inf`, and `Inf`.

In other words, none of your function's outputs should be an empty array.

Hints:

- You may want to consider using recursion for this question.
- If the variable "`c`" is a 1×1 array of class `char` *i.e.* a character string that contains only one character, you can check whether this character qualifies as "upper case" for this question by using the logical expression `c == upper(c)`. This logical expression will evaluate to true (logical 1) if the character should be considered upper case, and false (logical 0) otherwise.

Test cases:

```
>> [u, l, n, s] = my_cell_extractor({4, NaN, 6, 0, 'a', -1, 'B'})
u =
B
l =
a
n =
     4     6     0    -1
s =
   NaN

>> array = {[1, 4, 5], 's', 'ays', 'E', 10, NaN, {-1, '7hi'}};
>> [u, l, n, s] = my_cell_extractor(array)
u =
E7
l =
sayshi
n =
     1     4     5    10    -1
s =
   NaN

>> a = {'r', 'ec', NaN, [4, 1], 'I', 'L', 'Iursion', {'KE', {{exp(10000)}}}, 0};
>> [u, l, n, s] = my_cell_extractor(a)
u =
ILIKE
l =
recursion
n =
     4     1     0
s =
   NaN   Inf
```