Lab Assignment #5

Due 2/26/2016 at 4pm on bCourses

Some guidelines for successfully completing an E7 assignment:

- First convert the engineering problem into a function that can be effectively calculated.
  - The function should be written in pseudo-code with pencil and paper first.
- Program the function.
  - Translate the math and pseudo-code into MATLAB.
  - Then run the program and test it to see if its outputs match those specified by the function. If they do, compute the function on the input data and check that your answers make sense and are reasonable. If they do not, look for any mistakes in your math or in your coding.

The assignment will be partially graded by an autograder which will check the values of the required variables, so it is important that your variable names are exactly what they are supposed to be (variable names ARE case-sensitive). Instructions for submitting your assignment are included at the end of this document.

1 Algorithm Complexity and Big-O Notation

In this problem, you are presented with three sample MATLAB functions, `f(x)`, `g(x)`, and `myRecFactorial(n)`, shown below. Your job is to determine the complexity of each from the options below, expressed in "Big-O" notation. First examine the three functions below and select from options A–E. Then write a function with the following header:

```
function [answers] = myBigONotation()
```

where `answers` is a 1×3 `cell` array where each `cell` contains a single `char` specifying your answer (chosen from the options below) for the three sample functions. For example, if you were to pick answer B for `f(x)`, answer E for `g(x)`, and answer D for `myRecFactorial(n)`, you would define the output as `answers = { 'B', 'E', 'D' }`.

Choices (same list of choices for the three functions):

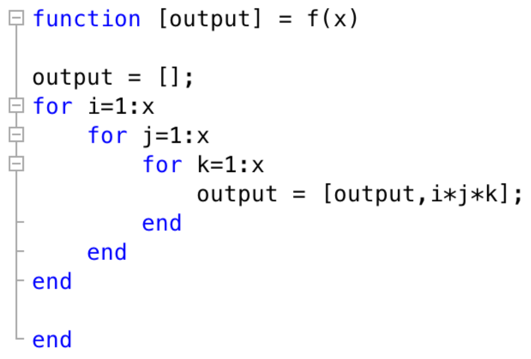
(A) $\mathcal{O}(n^2)$.

(B) $\mathcal{O}(n)$.

(C) $\mathcal{O}(C^n)$, where C is a constant.

(D) $\mathcal{O}(\log(n))$.

(E) $\mathcal{O}(n^3)$.

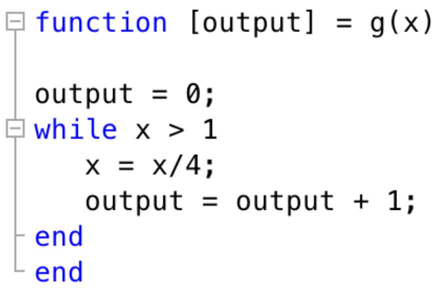


```
function [output] = f(x)

output = [];
for i=1:x
    for j=1:x
        for k=1:x
            output = [output,i*j*k];
        end
    end
end

end
```

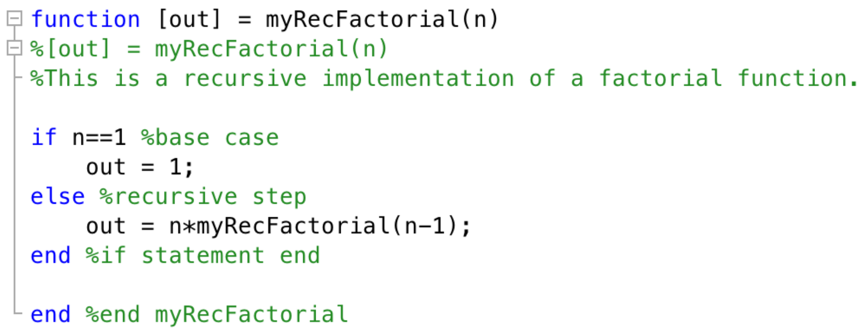
Figure 1: Function f



```
function [output] = g(x)

output = 0;
while x > 1
    x = x/4;
    output = output + 1;
end
end
```

Figure 2: Function g



```
function [out] = myRecFactorial(n)
%[out] = myRecFactorial(n)
%This is a recursive implementation of a factorial function.

if n==1 %base case
    out = 1;
else %recursive step
    out = n*myRecFactorial(n-1);
end %if statement end

end %end myRecFactorial
```

Figure 3: Function myRecFactorial

2 Braille

English Braille is a writing system that uses binary encoding. In grade 1 English Braille, each letter of the alphabet and each number are represented by a cell that contains raised dots at one or more of six locations, arranged as an array with 3 rows and 2 columns. Table 1 shows the English Braille representation of the 10 digits of the decimal system. We will represent a Braille cell in MATLAB as a 3×2 array where raised dots are represented by ones, and zeros are used elsewhere. We will represent a sequence of n Braille cells in MATLAB as a $3 \times 2n$ array where individual cells have been concatenated horizontally. For example, the integer 2 is represented as the 3×2 array: `[1, 0; 1, 0; 0, 0]` and 24 is represented as the 3×4 array: `[1, 0, 1, 1; 1, 0, 0, 1; 0, 0, 0, 0]`. Note that in English Braille, numbers are preceded by a "numeral sign" character, which we omit in this assignment.

Table 1: English Braille representation (the black dots indicate the raised dots) of the ten digits of the decimal system. Images are from: https://en.wikipedia.org/wiki/English_Braille

● ○	● ○	● ●	● ●	● ○	● ●	● ●	● ○	○ ●	○ ●
○ ○	● ○	○ ○	○ ●	○ ●	● ○	● ●	● ●	● ○	● ●
○ ○	○ ○	○ ○	○ ○	○ ○	○ ○	○ ○	○ ○	○ ○	○ ○
1	2	3	4	5	6	7	8	9	0

In this problem, you will write two MATLAB functions that convert numbers from their Braille representations into other representations.

2.1 Braille conversion to double

Write a function with the following header:

```
function [result] = myBraille2Double(braille)
```

where **braille** is a $3 \times 2n$ array which represents an integer as a sequence of Braille cells as described above, and **result** is the number (MATLAB class "double") represented by **braille**.

Test case:

```
>> result = myBraille2Double([1,0,1,1;1,0,0,1;0,0,0,0])

result =

    24

>> result = myBraille2Double([1,0,1,1,1,0;1,0,0,1,0,0;0,0,0,0,0,0])

result =

   241
>> result = myBraille2Double([1, 0, 0, 1, 1, 0; ...
                               0, 1, 1, 0, 1, 1; ...
                               0, 0, 0, 0, 0, 0])
```

```
result =

    598
```

2.2 Braille conversion to ASCII code

Write a function with the following header:

```
function [ASCII] = myBraille2ASCII(braille)
```

where `braille` is a $3 \times 2n$ array that represents an integer as a sequence of Braille cells as described above, and `ASCII` is a $1 \times n$ row vector, where each element is the ASCII code (MATLAB class "double") of the corresponding cell in `braille`. Table 2 shows the ASCII codes of the ten digits of the decimal system.

Table 2: A section of the ASCII table.

Character	0	1	2	3	4	5	6	7	8	9
ASCII code	48	49	50	51	52	53	54	55	56	57

In MATLAB, you can obtain the ASCII code of a character by using the following (for example with the char `'5'`) `double('5')` or `'5' + 0`. You can also check what character is generated by an ASCII code by using `char(53)` which will in this case return `5`. See http://www.asciitable.com/ for more information on ASCII tables.

3 Binary representations of integers

There are many ways to represent integers in binary format (i.e. with only zeros and ones). In this problem, we only consider 8-bit representations: each integer will be represented in binary format by a sequence of 8 zeros and/or ones. You will learn and use three different binary representations for integers: "unsigned representation", "sign-magnitude representation", and "two's complement representation". Each of these representations is detailed below.

In the **unsigned representation**, each bit represents a power of 2, from 2^0 (right-most bit) to 2^7 (left-most bit). For example, the binary representation 00100100 represents the integer $2^2 + 2^5 = 36$, and the 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 representation.

In the **signed-magnitude** 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 2^6 (seventh bit from the right). For example, the binary representation 00100100 still represents the number $2^2 + 2^5 = 36$, while the binary representation 11101000 now represents the number $-(2^6 + 2^5 + 2^3) = -104$. Note that there are two possible binary representations of zero in the sign-magnitude representation (00000000 and 10000000).

In the **two's complement** representation, the left-most bit represents -2^7 if it is 1, and 0 if it is 0. The other seven bits are used in the same way as in the unsigned and signed-magnitude representations. For example, the binary representation 00100100 still represents the number 36, while the binary representation 11101000 now represents the number $-2^7 + 2^6 + 2^5 + 2^3 = -24$.

Write a function with the following header:

```
function [result] = myBinary2Num(binary, representation)
```

where `binary` is a character string of length 8 made of only zeros and ones, and `representation` is the name of the representation used for `binary`, given as a character string that can take one of the three following values: `'unsigned'`, `'sign-magnitude'` and `'twos complement'`. Your function should return in its output argument `result` the number (in base 10, MATLAB class `double`) that is represented by `binary` using the binary representation specified by `representation`.

Hint: You may find MATLAB's built-in function `bin2dec` useful when implementing your function. `bin2dec` converts a binary string into the corresponding integer (MATLAB class `double`) using the unsigned binary representation. You can also use `bin2dec` to check your function's outputs for more test cases (but only when using the unsigned representation).

Test cases:

```
>> result = myBinary2Num('11001000', 'unsigned')

result =

    200

>> result = myBinary2Num('11001000', 'sign-magnitude')

result =

    -72

>> result = myBinary2Num('11001000', 'twos complement')

result =
   -56
```

4 Floating point numbers

4.1 Comparing floating point numbers

Write a function with the following header:

```
function [exact, approx] = myCompareFloats(x, y, tolerance)
```

where `x`, `y`, and `tolerance` are scalars of class `double`, and `exact` and `approx` are scalars of class `logical`. `exact` should be true (logical 1) if and only if `x == y` evaluates to true in MATLAB. `approx` should be true if and only if the difference between `x` and `y` (in absolute value) is less than or equal to `tolerance`. In other words, the function's output argument `approx` indicates whether `x` and `y` are "approximately equal", where "approximately equal" is quantified by the input argument `tolerance`.

Test cases:

```
>> [exact, approx] = myCompareFloats(2+3, 5, 0)

exact =

     1


approx =

     1

>> [exact, approx] = myCompareFloats(0, 0.001, 1e-2)

exact =

     0


approx =

     1

>> [exact, approx] = myCompareFloats(0, 0.001, 1e-9)
```

```
exact =

     0


approx =

     0

>> [exact, approx] = myCompareFloats(0.3, 0.3, 1e-6)

exact =

     1


approx =

     1

>> [exact, approx] = myCompareFloats(0.1+0.2, 0.3, 1e-6)

exact =

     0


approx =

     1
```

You do not have to submit any answer to the following questions to complete this assignment, but you should think about the answers to improve your understanding of the use of floating point numbers in MATLAB.

- Why is `exact` equal to `0` in the last test case, while the following equality is mathematically true: $0.1 + 0.2 = 0.3$? **Hints:** you can use `fprintf('\%.25f\n', x)` to display the value of variable `x` with 25 digits after the decimal point. Try displaying `0.3` and then `0.1+0.2` with 25 digits.
- Given two numbers x and y, how would you decide what value to use for the function's input argument tolerance?

4.2 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 consider single precision 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:

$$
\begin{array}{ll}
(-1)^s \times 2^{e-d} \times (1+f) & \text{if } e \neq 0 \text{ and } e \neq 255 \\
(-1)^s \times 2^{1-d} \times f & \text{if } e = 0 \text{ and } f \neq 0 \\
0 & \text{if } e = 0 \text{ and } f = 0 \\
(-1)^s \infty & \text{if } e = 255 \text{ and } f = 0 \\
\text{NaN (Not a Number)} & \text{if } e = 255 \text{ and } f \neq 0
\end{array}
$$

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 Problem 3); and
- The value of f is calculated using bits 10 through 32, each bit representing a *negative* power of 2, from 2^{-1} (10th bit) to 2^{-23} (32nd bit).

Write a function with the following header:

```
function [result] = mySingle2Decimal(binary)
```

where `binary` is a character string of length 32 made of only zeros and ones, and `result` is the floating point number (in base 10, MATLAB class "double") that is represented by `binary` using the IEEE-754 32-bit single precision binary representation. In MATLAB, use `Inf` as the infinite value and `NaN` for quantities which are ”not a number”. Hint! See the example at the bottom of page 127 in the textbook (Chapter 8).

Test cases:

```
>> result = mySingle2Decimal('00111111111100000000000000000000')

result =

    1.8750

>> result = mySingle2Decimal('10111111000000000000000000000000')

result =

   -0.5000

>> result = mySingle2Decimal('00100000100000000000000000000001')

result =

   2.1684e-19

>> result = mySingle2Decimal('11111111100000000000000000000000')

result =

  -Inf

>> result = mySingle2Decimal('11111111100000000000000000000001')

result =

   NaN
```

You do not have to submit any answer to the following questions to complete this assignment, but you should think about the answers to improve your understanding of the representation of floating point numbers in computers.

- What is the “precision” of the IEEE-754 representations when representing a number close to 0? What about for a number close to 10^{20}? In MATLAB use the function `eps` to investigate the precision of the IEEE-754 representation for a given number (example for zero: `eps(0)` for the IEEE-754 64-bit representation and `eps(single(0))` for the IEEE-754 32-bit representation).

- What is the trade-off for using either the IEEE-754 32-bit or the 64-bit representation?

5 Sorting and efficiency of sorting algorithms

In this problem, you will write MATLAB functions to sort elements of the periodic table in alphabetical order (in parts a and b) and then compare (in part c) the efficiency of your sorting function against the efficiency of another sorting function that is given to you on bCourses. Note that the grading will **not** depend on the efficiency of your sorting algorithm, but your function does need to sort the elements properly.

5.1 Compare two elements

Write a function with the following header:

```
function [result] = myCompareElements(element1, element2)
```

where `element1` and `element2` are the names (as character strings) of two elements of the periodic table (for example: `'Hydrogen'`, `'Carbon'`). The full list of the elements of the periodic table can be found in Figure 5, and in a separate text file on bcourses.

Your function should return one of the following numbers (MATLAB class "double"):

- 0 if `element1` and `element2` are the same element.
- 1 if the name of `element1` comes before the name of `element2` in alphabetical order.
- -1 if the name of `element1` comes after the name of `element2` in alphabetical order.

In this assignment, alphabetical ordering is determined by the ASCII codes of the letters making up the names of the elements. A letter with a small ASCII code comes before a letter with a larger ASCII code. Note that in this assignment, the names of the elements will always be given to you with the first letter in upper case and all the other letters in lower case. **Your function `myCompareElements` must not use MATLAB's built-in function `sort`** nor the function `GSISortElements` which is available on bCourses.

Hint: you can compare the alphabetical ordering of two **single** characters in MATLAB with the logical operators `<`, `>`, and `==`. For example, `'b' < 'a'` will evaluate to false (0).

Test cases:

```
>> element1 = 'Hydrogen'; element2 = 'Carbon';
>> result = myCompareElements(element1, element2)
```

```
result =

    -1

>> result = myCompareElements(element2, element1)

result =

     1

>> result = myCompareElements(element1, element1)

result =

     0
```

5.2 Sort a list of elements

Write a function with the following header:

```
function [sorted] = mySortElements(elements)
```

where `elements` is a $1 \times n$ cell array that represents a list of n elements of the periodic table. Each cell of the cell array contains the name of an element of the periodic table as a character string. Multiple cells of `elements` can represent the same element (i.e. there can be duplicate elements in the list).

The function should return a $1 \times n$ cell array that represents the same list of elements as the one represented by `elements` but sorted in alphabetical order. If `elements` is an empty cell array, then `sorted` should be an empty cell array. **Your function `mySortElements` must not use MATLAB's built-in function `sort` or the function `GSISortElements`** that is available on bCourses.

Hints:

- You may want to use the function `myCompareElements` which you wrote in part 1 of this problem.
- You can find below the description of a sorting algorithm that you can use (it is more simple to implement but less efficient than many other sorting algorithms):
  1. Create an empty cell array (e.g. `sorted={}`) that will eventually contain the sorted elements;

2. Find the element in the unsorted cell array that has the lowest rank in alphabetical order. (**Hint:** your function `myCompareElements` may be useful to achieve this goal);
3. Add that element at the end of the sorted cell array;
4. Remove that element from the unsorted cell array (Use `elements(i) = []` to remove the cell with index `i` from cell array `elements`);
5. Repeat steps 2 through 5 until the unsorted array is empty.

- You can use MATLAB's built-in function `sort` or the function `GSISortElements` (see next part) to check that your function is working properly. However, you cannot call these functions from within your function `mySortElements`.

Test cases:

```
>> elements = {'Hydrogen', 'Calcium'};
>> sorted = mySortElements(elements)

sorted =

    'Calcium'    'Hydrogen'
>> elements = {'Hydrogen', 'Carbon', 'Magnesium', 'Calcium', 'Carbon'};
>> sorted = mySortElements(elements)

sorted =

    'Calcium'    'Carbon'    'Carbon'    'Hydrogen'    'Magnesium'
```

5.3 Efficiency of sorting algorithms

You must complete part 2 of this problem before you can do this part. Write a function that compares the efficiency of the `mySortElements` function you wrote in part 2 of this problem to the efficiency of the sorting function `GSISortElements` that is available as a .p file on bCourses. Here, we define efficiency as the time taken by the sorting function to sort a given list of elements. Note that there are many other criteria that should be considered for a more exhaustive characterization of the efficiency of an algorithm (for example RAM usage), but we ignore these in this assignment. Use the following header for your function:

```
function [] = myCompareSorting(n)
```

The function's input argument `n` is an integer (MATLAB class `double`), strictly greater than zero. Your function must create a plot of the time taken by your function `mySortElements` (as well as the time taken by the function `GSISortElements`) to sort an array of elements versus the size of the array. More precisely, your function must do the following:

- For each integer m between 1 and n (1 and n included):
  - Generate a $1 \times m$ cell array of randomly chosen elements of the periodic table. Use the function `GSIRandomElements` available as a .p file on bCourses to generate such a cell array. The header of the `GSIRandomElements` function is `function [elements] = GSIRandomElements(m)`, where `m` is an integer (MATLAB class `double`) larger than or equal to zero, and `elements` is a $1 \times m$ cell array of randomly chosen elements of the periodic table.
  - Sort the same list of elements twice, once using the `mySortElements` function you wrote in part 2 and once more using the function `GSISortElements`. The input and output arguments of `GSISortElements` are the same as for `mySortElements`. Measure the amount of time taken by these two functions (separately) to sort the list of elements using the MATLAB functions `tic` and `toc`. Here we call these times "execution times".
  - During execution of the function `myCompareSorting` the function `GSIRandomElements` should only be called once to create a $1 \times n$ cell array. To sort a smaller ($1 \times m$) array, use the first m elements of the output of `GSIRandomElements`.
- On a single figure, plot the execution times corresponding to your `mySortElements` function versus the number of elements being sorted. On the same plot, use another color to plot the execution times corresponding to `GSISortElements` function versus the number of elements being sorted. Format the plot appropriately (i.e. label x- and y-axes, add a legend box or line labels, etc.). Figure 4 shows an example obtained with two different sorting algorithms and $n = 200$. You can use a legend box instead of line labels on your plot.

Hint: You can find below an example of the use of the MATLAB functions `tic` and `toc` to measure the execution time of a piece of code:

```
tic;
F = FibRec(25);
time = toc;
```

After executing this code, the variable `time` will contain the execution time (in seconds) of the piece of code located between the calls to `tic` and `toc`, which in this case is a call to the function `FibRec`.

Note that the grading will **not** depend on the efficiency of your sorting algorithm (but your function `mySortElements` must sort the elements properly). For this problem, it does not matter whether your function `mySortElements` runs slower or faster than the sample function `GSISortElements`. One goal of this problem is to confront you with the fact that there might be multiple ways to achieve the same goals (e.g. sort in alphabetical order) using computer programming. Some algorithms are more efficient than others, some algorithms are easier to implement than others, etc. Note that the efficiency of a given algorithm can

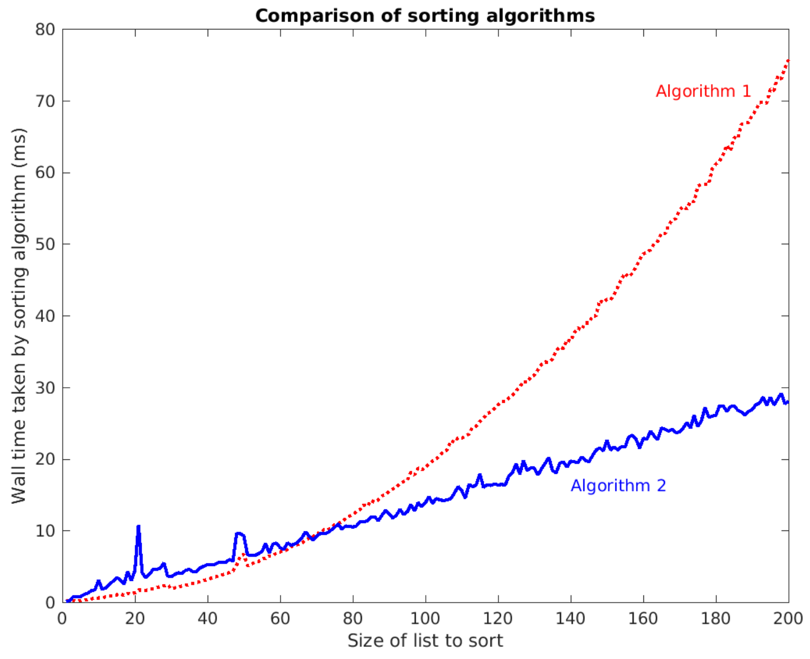


Figure 4: Efficiency of two different sorting algorithms.

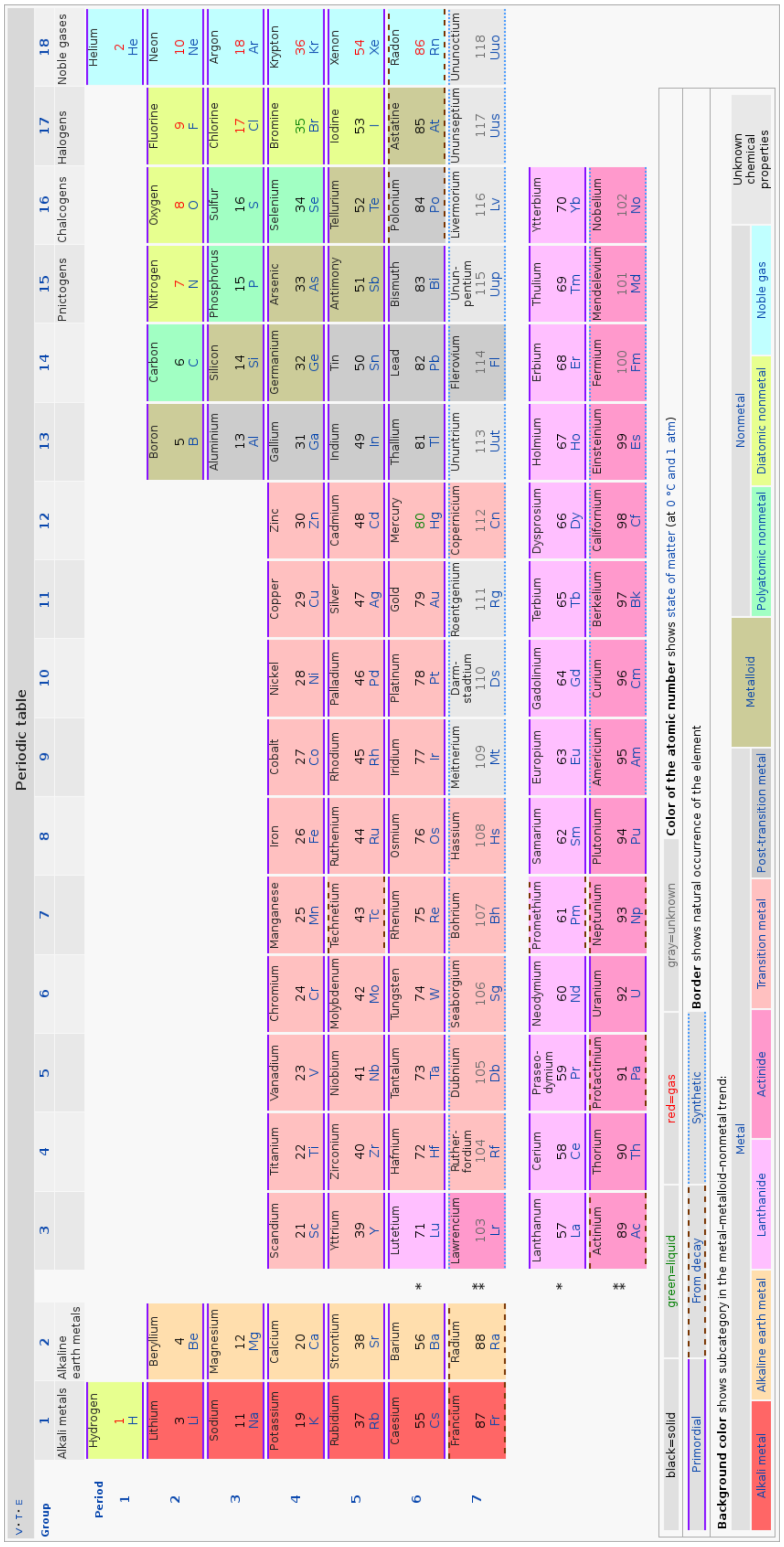
vary under different conditions (e.g. is the list of elements already a little bit or completely sorted?). Also note that execution time depends on other criteria such as computer hardware, amount of RAM used by other applications, etc. You do **not** need to submit the plot for this problem, just the function file.

Submission instructions

Your submission should include the following function files in a single zip file named FIRSTNAME_LASTNAME_LAB5.zip

- myBigONotation.m
- myBraille2Double.m
- myBraille2ASCII.m
- myBinary2Num.m
- myCompareFloats.m
- mySingle2Decimal.m
- myCompareElements.m
- mySortElements.m
- myCompareSorting.m

Don't forget that the function headers you use must appear *exactly* as they do in this problem set. So, make sure the variables have the right capitalization, the functions have the correct name, and the inputs and outputs are in the correct order.



V · T · E

Periodic table

Group →	1 Alkali metals	2 Alkaline earth metals		3	4	5	6	7	8	9	10	11	12	13	14	15 Pnictogens	16 Chalcogens	17 Halogens	18 Noble gases
Period 1	Hydrogen 1 H																		Helium 2 He
2	Lithium 3 Li	Beryllium 4 Be												Boron 5 B	Carbon 6 C	Nitrogen 7 N	Oxygen 8 O	Fluorine 9 F	Neon 10 Ne
3	Sodium 11 Na	Magnesium 12 Mg												Aluminium 13 Al	Silicon 14 Si	Phosphorus 15 P	Sulfur 16 S	Chlorine 17 Cl	Argon 18 Ar
4	Potassium 19 K	Calcium 20 Ca		Scandium 21 Sc	Titanium 22 Ti	Vanadium 23 V	Chromium 24 Cr	Manganese 25 Mn	Iron 26 Fe	Cobalt 27 Co	Nickel 28 Ni	Copper 29 Cu	Zinc 30 Zn	Gallium 31 Ga	Germanium 32 Ge	Arsenic 33 As	Selenium 34 Se	Bromine 35 Br	Krypton 36 Kr
5	Rubidium 37 Rb	Strontium 38 Sr		Yttrium 39 Y	Zirconium 40 Zr	Niobium 41 Nb	Molybdenum 42 Mo	Technetium 43 Tc	Ruthenium 44 Ru	Rhodium 45 Rh	Palladium 46 Pd	Silver 47 Ag	Cadmium 48 Cd	Indium 49 In	Tin 50 Sn	Antimony 51 Sb	Tellurium 52 Te	Iodine 53 I	Xenon 54 Xe
6	Caesium 55 Cs	Barium 56 Ba	*	Lutetium 71 Lu	Hafnium 72 Hf	Tantalum 73 Ta	Tungsten 74 W	Rhenium 75 Re	Osmium 76 Os	Iridium 77 Ir	Platinum 78 Pt	Gold 79 Au	Mercury 80 Hg	Thallium 81 Tl	Lead 82 Pb	Bismuth 83 Bi	Polonium 84 Po	Astatine 85 At	Radon 86 Rn
7	Francium 87 Fr	Radium 88 Ra	**	Lawrencium 103 Lr	Ruther-fordium 104 Rf	Dubnium 105 Db	Seaborgium 106 Sg	Bohrium 107 Bh	Hassium 108 Hs	Meitnerium 109 Mt	Darm-stadtium 110 Ds	Roentgenium 111 Rg	Copernicium 112 Cn	Ununtrium 113 Uut	Flerovium 114 Fl	Unun-pentium 115 Uup	Livermorium 116 Lv	Ununseptium 117 Uus	Ununoctium 118 Uuo
			*	Lanthanum 57 La	Cerium 58 Ce	Praseo-dymium 59 Pr	Neodymium 60 Nd	Promethium 61 Pm	Samarium 62 Sm	Europium 63 Eu	Gadolinium 64 Gd	Terbium 65 Tb	Dysprosium 66 Dy	Holmium 67 Ho	Erbium 68 Er	Thulium 69 Tm	Ytterbium 70 Yb		
			**	Actinium 89 Ac	Thorium 90 Th	Protactinium 91 Pa	Uranium 92 U	Neptunium 93 Np	Plutonium 94 Pu	Americium 95 Am	Curium 96 Cm	Berkelium 97 Bk	Californium 98 Cf	Einsteinium 99 Es	Fermium 100 Fm	Mendelevium 101 Md	Nobelium 102 No		

Color of the atomic number shows state of matter (at 0 °C and 1 atm)

black=solid	green=liquid	red=gas	gray=unknown

Border shows natural occurrence of the element

Primordial	From decay	Synthetic

Background color shows subcategory in the metal–metalloid–nonmetal trend:

Metal						Metalloid	Nonmetal			Unknown chemical properties
Alkali metal	Alkaline earth metal	Lanthanide	Actinide	Transition metal	Post-transition metal		Polyatomic nonmetal	Diatomic nonmetal	Noble gas	

Figure 5: Periodic table of the elements. Source: https://en.wikipedia.org/wiki/Periodic_table