System Programming in C – Homework Exercise 5

Publication date: *Wednesday ,June 10, 2020*
Due date: *Sunday, June 28, 2020* @ 21:00

General overview:

- The purpose of this assignment is to implement a program that sums up a list of non-negative numbers using **dynamic precision**. Standard number types, such as **`int`** or **`double`** have pre-defined precision, meaning that they can accurately represent a certain collection of number values. In this assignment you will address this problem by defining a new type of number represented using a string. The use of strings will allow us to represent numbers in any precision we wish.
- In Problem 1, you will implement a series of functions on **number strings** in file `numberString.c`, and in Problem 2, you will write a **`main()`** function for the program in a separate file named `sumNumbers.c`. Initial versions of both files are provided in `/share/ex_data/ex5/`.

Coding and compilation guidelines:

- Follow the implementation guidelines specified for each function. Not all guidelines are covered by the automatic tests, but they will be checked manually by the graders, so **make sure to stick to the guidelines**.
- Write clear and readable code. Use appropriate indentation and try to follow the style of `numberString.c` as much as possible. You should also add brief documentation for the critical parts of each function's implementation. Keep in mind that your code is reviewed by the graders and 10 grade points will be allocated for code style.
- Make sure that your code compiles without errors or warnings and passes all the tests specified for each task. Note that Problem 1 and Problem 2 have different compilation instructions.

Problem 1:

The purpose of this question is to implement seven functions that operate on strings that represent numbers, which we refer to as "**number strings**". A string represents a number if it consists of a sequence of decimal digits (characters `'0'-'9'`), and possibly one decimal point (`'.'`). In particular, a number string contains at most one decimal point, and all other characters are digit characters. Furthermore, a number string cannot be empty and its first character is a digit (not the decimal point). Note that in this setting, we do not use sign characters (`'+'` or `'-'`), and we do not consider negative numbers.

The following table specifies several examples of number strings and their value:

Number string	**Value**
`"3040.203"`	3040.203
`"003040.203000"`	3040.203
`"3040.0"`	3040
`"03040."`	3040
`"3040"`	3040
`"0.3040"`	0.304

Note that a number string may contain **leading zeros** on its left end, as in examples `"003040.203000"` and `"03040."`, and/or **trailing zeros** on its right end, as in examples `"003040.203000"`, `"3040.0"`, and `"0.3040"`. See Problem 1.2 and 1.3 below for more details on leading and trailing zeros. The following strings are not number strings:

String	**Why is this not a number string?**
`"-3040.203"`	`'-'` is not a digit or a period character
`"3040.203."`	More than one decimal point
`"3.040e3"`	`'e'` s not a digit or a period character
`".304"`	Decimal point cannot be first character
`""`	Number strings must contain at least one digit

Coding guidelines:

- Copy over the source file `numberString.c` from the shared directory `/share/ex_data/ex5/` to `~/exercises/ex5/`. In each of the seven tasks below you are required to complete the implementation of a single function.
- Follow the instructions carefully, and make sure to modify the code of `numberString.c` only in the marked space.
- In each task you are allowed to invoke functions from the standard library `string.h` and functions you implemented in previous tasks. You are encouraged to do this when possible. Do not make use of functions from other standard C libraries (such as `stdio`).

Compiling and testing your code:

- You should test your solution to each task below by compiling your code with the test code file `/share/ex_data/ex5/test_ex5.c`. This file contains a main function, which runs tests for every function you implement. You "turn on" a specific test by compiling your source file together with the `test_ex5.c` and using the `-D TEST_<QUESTION_ID>` option, where `<QUESTION_ID>` is the question id. Example for testing (`<QUESTION_ID>` is set to `1_4` for Problem 1.4):

```
==> gcc -Wall -D TEST_1_4 \
        /share/ex_data/ex5/test_ex5.c numberString.c \
        -o test_ex5_1_4
```

  [We discuss compilation of multi-file programs in detail in weeks #11 and #12]

- After your source has been **successfully compiled** (no errors or warning), you should run the program (`test_ex5_<QUESTION_ID>`) and compare the output with the expected output provided in file `test_ex5_<QUESTION_ID>.out` in the shared directory `/share/ex_data/ex5/`:

```
==> ./test_ex5_1_4 > test_ex5_1_4.out
==> diff /share/ex_data/ex5/test_ex5_1_4.out test_ex5_1_4.out
```

1. Complete the implementation of the function **`decimalPointPosition`**, such that it identifies whether a given string (**`const char* numStr`**) is a valid number string, and if so, it returns the position of the decimal point. For a complete definition of valid number strings, see page 2. Follow these guidelines:
   - If **`numStr`** is a valid number string with a decimal point, the function should return its position within the string (using 0-based indexing).

- If **`numStr`** is a valid number string without a decimal point, the function should return the length of the string. Note that such number strings represent integers, so we implicitly "place" the decimal point after the number.
- If **`numStr`** is not a valid number string, the function should return -1.

Execution examples:

- **`decimalPointPosition("3040.203")`** returns 4
- **`decimalPointPosition("003040.203000")`** returns 6
- **`decimalPointPosition("03040.")`** returns 5
- **`decimalPointPosition("3040")`** returns 4
- **`decimalPointPosition("3040.203.")`** returns -1
- **`decimalPointPosition("3.040e3")`** returns -1
- **`decimalPointPosition(".304")`** returns -1
- **`decimalPointPosition("")`** returns -1

Validation: Compile your code and validate it according to the guidelines on page 3, using question label `<QUESTION_ID>=1_1`.

2. Complete the implementation of the function **`numLeadingZeros`**, such that it returns the number of leading zeros in a given number string (**`const char* numStr`**), according to the following guidelines:
   - If **`numStr`** is a valid number string, the number of leading zeros is the length of the longest prefix (רישא) of the string that consists only of zero digits (`'0'`) and not including the units digit (the one immediately left of the dec. point).
   - If **`numStr`** is not a valid number string, the function should return -1.

   Execution examples:

   - **`numLeadingZeros("3040.203")`** returns 0
   - **`numLeadingZeros("003040.203000")`** returns 2
   - **`numLeadingZeros("03040.")`** returns 1
   - **`numLeadingZeros("000.340")`** returns 2
   - **`numLeadingZeros("0.34000")`** returns 0
   - **`numLeadingZeros("000")`** returns 2
   - **`numLeadingZeros(".304")`** returns -1
   - **`numLeadingZeros("")`** returns -1

   Validation: Compile your code and validate it according to the guidelines on page 3, using question label `<QUESTION_ID>=1_2`.

3. Complete the implementation of the function **`numTrailingZeros`**, such that it returns the number of trailing zeros in a given number string (**`const char* numStr`**), according to the following guidelines:

- If **`numStr`** is a valid number string, the number of trailing zeros is the length of the longest suffix (סיפא) of the string that consists only of zero digits (`'0'`) and is to the right of the decimal point.
- If **`numStr`** is not a valid number string, the function should return -1.

Execution examples:

- **`numTrailingZeros("3040.203")`** returns 0
- **`numTrailingZeros("003040.203000")`** returns 3
- **`numTrailingZeros("03040.")`** returns 0
- **`numTrailingZeros("000.340")`** returns 1
- **`numTrailingZeros("0.00000")`** returns 5
- **`numTrailingZeros("000")`** returns 0
- **`numTrailingZeros(".304")`** returns -1
- **`numTrailingZeros("")`** returns -1

Validation: Compile your code and validate it according to the guidelines on page 3, using question label `<QUESTION_ID>=1_3`.

4. Replace the return statement in function **`numStringIsInteger`**, such that it returns 1 if its parameter (**`const char* numStr`**) corresponds to a number string with integer value, and 0 otherwise (if it is a number string with non-integer value or not a number string).Your implementation should only replace the return value (-123) with the appropriate one-line expression. The return expression may use the local variable **`decPos`**, various operators, and function invocations.

   Execution examples:

   - **`numStringIsInteger("304")`** returns 1
   - **`numStringIsInteger("03040.")`** returns 1
   - **`numStringIsInteger("111.000")`** returns 1
   - **`numStringIsInteger("0.00000")`** returns 1
   - **`numStringIsInteger("3040.203")`** returns 0
   - **`numStringIsInteger("000.340")`** returns 0
   - **`numStringIsInteger(".304")`** returns 0
   - **`numStringIsInteger("")`** returns 0

   Validation: Compile your code and validate it according to the guidelines on page 3, using question label `<QUESTION_ID>=1_4`.
   Manual inspection: make sure that you only replace the return statement and have no other statement in your code.

5. Complete the implementation of the function **`numStringToDouble`**, such that it returns the **`double`** value of a given number string (**`const char* numStr`**). If **`numStr`** is not a number string, the function should return -1.0. You may find the implementation of function **`atoi()`** useful (see Lecture/Recitation #9).

Execution examples:

- `numStringToDouble("304")` returns 304.0
- `numStringToDouble("03040.")` returns 3040.0
- `numStringToDouble("3040.203")` returns 3040.203
- `numStringToDouble("000.340")` returns 0.34
- `numStringToDouble("000.000")` returns 0.0
- `numStringToDouble(".304")` returns -1.0
- `numStringToDouble("")` returns -1.0
- `numStringToDouble("3.0000000000000001") - 3` returns 0.0
- `numStringToDouble("3.00000000000001") - 3` returns 1e-14

Explanation of the last two examples: the relative precision of type **`double`** allows it to save 16 decimal digits. Our number strings can be more precise that **`double`** because they are not restricted to a pre-defined number of allocated bytes.

Validation: Compile your code and validate it according to the guidelines on page 3, using question label `<QUESTION_ID>=1_5`.

6. Complete the implementation of the function **`standardizeNumString`**, such that it standardizes a given number string (**`char* numStr`**) by removing all leading and trailing zeros, according to the following guidelines:
   a. The standard version of a given number string is a number string that represents the same value but does not have leading or trailing zeros.
   b. Furthermore, the standard version of a number string that represents an integer does not have a decimal point.
   c. The function modifies the inputted **`numStr`** with the standard version of the number string. Note that the input number string is at least as long as the standardized version that the function writes as output.
   d. Make sure to write the terminating `'\0'` in the appropriate position, and do not modify characters in the input string beyond this position.
   e. If **`numStr`** is not a valid number string, then the function should not modify the input string and return 0.
   f. If **`numStr`** was already a standard number string, then the function should not modify the input string and return 1.
   g. If **`numStr`** is a non-standard valid number string, then the function should standardize the number string as specified above, and then return 0.

Execution examples:

- **`standardizeNumString("0001020304050.050403020100")`** returns value **0** and modifies string to **`"1020304050.0504030201".`**
  (leading zeros and trailing zeros removed)
- **`standardizeNumString("0001020304050.000000")`** returns value **0** and modifies string to **`"1020304050".`**
  (leading zeros, trailing zeros and decimal point removed)
- **`standardizeNumString("0001020304050.")`** returns value **0** and modifies string to **`"1020304050".`**
  (leading zeros and decimal point removed)
- **`standardizeNumString("0012345678900")`** returns value **0** and modifies string to **`"12345678900".`**
  (leading zeros removed)
- **`standardizeNumString("0000000000000000000")`** returns value **0** and modifies string to **`"0".`**
  (leading zeros removed – "unit 0" is not leading)
- **`standardizeNumString("0000000000.01000000")`** returns value **0** and modifies string to **`"0.01".`**
  (leading zeros and trailing zeros removed – "unit 0" is not leading)
- **`standardizeNumString("0")`** returns value **1** and modifies string to **`"0".`**
  (already standard – nothing to do)
- **`standardizeNumString("12300450.0670089")`** returns value **1** and modifies string to **`"12300450.0670089".`**
  (already standard – nothing to do)
- **`standardizeNumString("00010000..00000")`** returns value **0** and modifies string to **`"00010000..00000".`**
  (not a valid number string)

Note: all examples assume that the input string is written in some buffer. Our tests also check that characters in the buffer after the terminating `'\0'` of the modified number string are not modified (see d above).

Validation: Compile your code and validate it according to the guidelines on page 3, using question label `<QUESTION_ID>=1_6`.
Manual inspection: make sure that when the input is not a valid number string, your function does not write anything and when it is a valid number string, your function does not write beyond the terminating `'\0'` of the standardized number string.

7. Complete the implementation of the function **`numStringSum`**, such that it computes the sum of two number strings as a new number string. The function receives as parameters: two number strings (**`const char *numStr1, const char *numStr2`**), a pointer to a buffer where the output should be written (**`char *numStrSum`**), and the maximum number of characters that the function is allowed to write in the buffer (**`int buffSize`**). The function computes the sum of the number strings contained in **`numStr1`** and **`numStr2`**, and it writes it in the buffer **`numStrSum`** if the sum corresponds to a number string with at most **`buffSize`**-1 characters (**`buffSize`** characters including the terminating `'\0'`). The function should follow the procedure outlined below.

   Summing two numbers is done by aligning them (one on top of the other), such that their decimal points are aligned. The **overlap** between the aligned numbers is then identified as the sequence of digits from each number that are aligned to digits in the other number. The unaligned digits may form up to two "**overhangs**" on the left and right of the overlap. In the examples below we demonstrate the alignment of four pairs, emphasizing the overlap in bold:

   - Alignment of "`3040.2`" and "`12.907`":
     (left overhang: `30` right overhang: `07`)

     ```
     30 40.2
       12.9 07
     ```
   - Alignment of "`3040.2`" and "`047`":
     (left overhang: `3` right overhang: `.2`)

     ```
     3 040.2
       047
     ```
   - Alignment of "`3040.20`" and "`0012.9`":
     (left overhang: none right overhang: `0`)

     ```
     3040.2 0
     0012.9
     ```
   - Alignment of "`12.9`" and "`304.2`":
     (left overhang: `3` right overhang: none)

     ```
       12.9
     3 04.2
     ```

   After the alignment of the two number strings is determined, their sum is computed by summing aligned pairs of digits from right to left. When summing a pair of digits there may be a "carry-over" from one position to the next (if the sum is greater than 9). If there is a carry-over from the left-most pair of digits in the overlap, the carry-over should continue to the left overhang. If the carry-over continues beyond the left-most digit in the left overhang, then there is an overflow of one digit. For instance, there is digit overflow in the sum of the following pairs: "`32`" with "`84.4`" (sum is "`116.6`") and "`999.4`" with "`3`" (sum is "`1002.4`").

Your implementation should follow these guidelines:

- If one of the input strings is not a valid number string, the function should not modify the contents of the buffer **`numStrSum`**, and return 0.
- Before the function writes anything into the buffer **`numStrSum`**, it should compute the expected length of the sum. This depends on the length of the overlap and overhangs, as defined above, and may include a single digit overflow (see next).
- The function should consider an overflow digit whenever the left-most digit in the overhang is 9, or the overhang is empty and the sum of the left-most aligned pair of digits is at least 9. Note that these are necessary conditions for an overflow to occur, but **they are not sufficient**. This means that there may be cases where the function leaves space for an overflow digit, but then it writes a leading 0 in that space (see execution examples on next page).
- If the expected length of the sum, including the possible overflow (see above) and the terminating `'\0'`, is **`len`** and **`len > buffSize`**, the function should not modify the contents of the buffer **`numStrSum`**, and return **`-len`**. Thus, a negative return value indicates that the buffer size is insufficient, and it also indicates the required buffer size for the sum.
- If both input strings are valid number strings and the expected length of the sum is **`len ≤ buffSize`**, then the function should write the sum into the buffer **`numStrSum`**, and return 1.
- The number string written into **`numStrSum`** does not have to be standardized (see execution examples).
- Make sure to terminate the string properly, and if there is an expected overflow, write the appropriate overflow digit (0 or 1; see execution examples on next page).

Important implementation note: write the sum directly into the provided buffer **`numStrSum`**. Do not define local arrays in **`numStringSum()`**, and do not use dynamic memory allocation. You may, however, use free pointers that point to locations within the input strings and/or the output buffer.

Execution examples:

- **`numStringSum("12313", "67859", buffer, 20)`** returns **`1`**
  and writes **`"80172"`** in the buffer.
  (no overhangs and several carry-overs)
- **`numStringSum("14342.0000", "67658", buffer, 20)`** returns **`1`**
  and writes **`"82000.0000"`** in the buffer.
  (right overhang in **`numStr1`** and no standardization)
- **`numStringSum("263.73", "678.521", buffer, 20)`** returns **`1`**
  and writes **`"942.251"`** in the buffer.
  (right overhang in **`numStr2`**)

- **`numStringSum("52313.7", "1009967859.6", buffer, 20)`** returns **1**
  and writes **`"1010020173.3"`** in the buffer.
  (left overhang in **`numStr2`** and carry-over into the overhang)
- **`numStringSum("8899", "01.99999", buffer, 20)`** returns **1**
  and writes **`"8900.99999" in the buffer.`**
  (left overhang in **`numStr1`** and right overhang in **`numStr2`**)
- **`numStringSum("9.876543210123456789", "8765.4321", buffer, 30)`**
  returns **1** and writes **`"8775.308643210123456789" in the buffer.`**
  (left overhang in **`numStr2`** and right overhang in **`numStr1`** and accuracy that cannot be represented using **`double`**)
- **`numStringSum("90", "1.234567", buffer, 20)`** returns **1**
  and writes **`"091.234567"`** in the buffer.
  ("unused" overflow digit **`0`** because first digit in left overhang is **`9`**)
- **`numStringSum("9.9901", "90.01", buffer, 20)`** returns **1**
  and writes **`" 100.0001"`** in the buffer.
  ("used" overflow digit **`1`** because first digit in left overhang is **`9`**)
- **`numStringSum("4", "5", buffer, 20)`** returns **1**
  and writes **`"09"`** in the buffer.
  ("unused" overflow digit **`0`** because sum of left-most digits in overlap is ≥ **`9`**)
- **`numStringSum("73", "27.1234", buffer, 20)`** returns **1**
  and writes **`"100.1234"`** in the buffer.
  ("used" overflow digit **`1`** because sum of left-most digits in overlap is ≥ **`9`**)
- **`numStringSum(".73", "27", buffer, 1)`** returns **0**
  and buffer contents are **`"--- buffer reset ---"`**.
  (**`numStr1`** is an invalid number string; **`"--- buffer reset ---"`** is the string written in buffer before calling **`numStringSum`**)
- **`numStringSum("73", "seven", buffer, 20)`** returns **0**
  and buffer contents are **`"--- buffer reset ---"`**.
  (**`numStr2`** is an invalid number string)
- **`numStringSum("9.9901", "90.01", buffer,  8)`** returns **-9**
  and buffer contents are **`"--- buffer reset ---"`**.
  (**`buffSize`** is insufficient – return minus the required buffer size)
- **`numStringSum("9.9901", "90.01", buffer, -1)`** returns **-9**
  and buffer contents are **`"--- buffer reset ---"`**.
  (**`buffSize`** is insufficient – return minus the required buffer size)

Validation: Compile your code and validate it according to the guidelines on page 3, using question label `<QUESTION_ID>=1_7`.

Manual inspection: make sure that your function does not use any locally allocated arrays or dynamically allocated arrays. Writing should be done only in the output buffer **`numStrSum`**.

Final testing for Problem 1: Execute script `/share/ex_data/ex5/test_ex5.1` from the directory containing your `numberString.c` source file.

(the script produces a detailed error report to help you debug your code)

Problem 2:

Write a **`main()`** function for a program that receives a list of number strings as arguments, and computes and prints their sum. The **`main()`** function should be written in a separate file named `sumNumbers.c`. Copy over an initial version of this file from `/share/ex_data/ex5/`. Do not add any `#include` directive to this initial version, but other than that, you may make any change you wish to this file. Implement **`main()`** according to the guidelines below.

Note on header files (.h): The file `sumNumbers.c` contains `#include` directives to four header files. header files simply contain declarations of function implemented in a separate source file (`.c`). Their inclusion allows you to call functions that are defined in other source files by satisfying the requirement that each function is declared before it is invoked. We discuss this in detail in the last two weeks of the course (as well as other topics related to multi-file programs).

Program specification:

- Your program receives a list of number strings as command line arguments.
- If an argument is not a valid number string, it is skipped.
- The program should print a single line in the following format:
  `The sum of the <N> input numbers is <SUM_NUMBER_STRING>`
  Where `<N>` is the number of valid number strings provided as input arguments and `<SUM_NUMBER_STRING>` is their sum, written as a standard number string.
- The exit status of the program (return value of **`main()`**) should be the number of invalid input arguments. Thus, the program exits with status 0 if execution completed successfully and all arguments are valid number strings.

Implementation guidelines:

- You are encouraged to call functions that you implemented in Problem 1 in your code for **`main()`**. To enable this, we added an `#include` directive to the header file `/share/ex_data/ex5/numberString.h` at the top of the initial version of `sumNumbers.c` (see note on header files).
- Do not convert the number string to a number value (**`double`**, **`int`**, etc.) at any stage. All calculations should be done on number strings.
- Standardize each number string before you incorporate it into the sum, to minimize the space required for holding the result.
- Standardize the resulting sum in each stage, since the sum of two standard number strings may have trailing zeros (for example, 0.4+0.6=1.0).
- Because the size of the buffer required to hold the sum is unknown in advance, you should use a dynamically allocated array to hold the sum.
- For dynamic memory allocation, you should only use the functions **`ourMalloc()`** and **`ourFree()`**, which we provide in the `virtualHeap` library. These functions should be applied exactly like **`malloc()`** and **`free()`**, and they allow us to inspect your memory allocation when testing your code. To enable use of these functions, we added an `#include` directive to the header file `/share/ex_data/ex5/virtualHeap.h` at the top of the initial version of

`sumNumbers.c` (see note on header files). Do not use the standard dynamic allocation functions from `stdlib.h`.

- Make sure to check that allocation succeeded before you use (de-reference) any dynamically allocated space. Our virtual heap function **`ourMalloc()`** returns NULL if it is not able to allocate the required amount of space. If any of your allocations do not succeed, you should print the following message and return an exit status of -1 (return value of main is -1):
  `Out of memory`
- In each iteration you should re-allocate space for the sum according to the exact number of bytes required to hold it (before standardization). You may obtain this size by "falsely" calling the function **`numStringSum()`**, with parameter **`buffSize=0`**. When invoked this way, the function should return the required buffer size (with negative sign). Use this size to allocate a new buffer every iteration, and make sure to free any dynamically allocated space if you are no longer using it.
- Before your return from **`main()`**, you should make sure to free any dynamically allocated memory that has not already been freed.
- Other than number string and virtual heap functions, you are also allowed to call functions from the standard libraries `stdio` and `string`. To enable this, we added `#include` directives to the header files `stdio.h` and `string.h` at the top of the initial version of `sumNumbers.c` (see note on header files).
- Do not add any more `#include` directives to `sumNumbers.c` (particularly not to `stdlib.h`).
- You may implement additional functions in `sumNumbers.c` and call them from **`main()`**, if you find this helpful. If you choose to do this, make sure to declare these functions appropriately.

Compilation guidelines and testing:

- To create the executable for your program, you should compile your code as follows:

```
==> gcc -Wall \
       sumNumbers.c numberString.c /share/ex_data/ex5/virtualHeap.c \
       -o sumNumbers
```

- This creates an executable program called `sumNumbers`, which you can run and test on basic execution examples provided on the next page, as well as more more advanced examples to test end cases.
- To help you test your code, we provide a working executable `/share/ex_data/ex5/sumNumbers`. You should prepare a set of inputs and compare your output with the output of our program. You may find a list of suggested inputs in `/share/ex_data/ex5/test_ex5.3.inputs.txt`. To try out the i^{th} input in this file, use this sequence of commands, (demonstrated here for i=2):

```
==> i=2
==> input_line=`cat /share/ex_data/ex5/test_ex5.2.inputs.txt| \
      grep -v "^#" | grep [^[:space:]] | head -n$i | tail -n1`
==> ./sumNumbers $input_line
==> /share/ex_data/ex5/sumNumbers $input_line
```

Basic execution examples:

```
==> ./sumNumbers
The sum of the 0 input numbers is 0
==> echo $?
0
```

← recall that **$?** holds the exit status of last program executed

```
==> ./sumNumbers 03405.045063000
The sum of the 1 input numbers is 3405.045063
==> echo $?
0

==> ./sumNumbers 1234567890123456 0.00.33 0.654321987654321
The sum of the 2 input numbers is 1234567890123456.654321987654321
==> echo $?
1

==> ./sumNumbers hello world how are you -5 .006
The sum of the 0 input numbers is 0
==> echo $?
7

==> ./sumNumbers hello 1 world 200 how 0.004 are 50 you
The sum of the 4 input numbers is 251.004
==> echo $?
5
```

Final testing for Problem 2: Execute script `/share/ex_data/ex5/test_ex5.2` from the directory containing your `numberString.c` and `sumNumbers.c` source files. The script compares the output of your program to the output of our program on a list of inputs, using the procedure described above. As always, it produces a detailed error report to help you debug your code).

Submission Instructions:

1. After you validated and tested your solution, make sure that your `~/exercises/ex5/` directory contains the following C source file, which includes your implementation:
   - `numberString.c`
   - `sumNumbers.c`
2. your `~/exercises/ex5/` directory should also contain a `PARTNER` file with the user id of the non-submitting partner. The non-submitting partner should also add a `PARTNER` file containing the user id of the submitting partner.
3. Check your solution by running **`check_ex ex5`**. The script should be executed from the account of the submitting partner, and it may be run from any directory. Clean execution of this script guarantees you 80% of the assignment's grade.
4. Once you are satisfied with your solution, you may submit it by running **`submit_ex ex5`**. The script should be executed from the account of the submitting partner, and it may be run from any directory. You may modify your submission any time before the deadline (**28/6 @ 21:00**) by running **`submit_ex ex5 -o`** from any location.
5. For more information on the submission process, see the **Homework submission instructions** file on the course website.