Problem Statement

Develop a C library for the integers of arbitrary length (intal).

intal

An intal is a nonnegative integer of arbitrary length, but it is sufficient for your implementation to support up to 1000 decimal digits. The integer is stored as a null-terminated string of ASCII characters. An intal is represented as a string of decimal digits ('0' thru '9') that are stored in the big-endian style. That is, the most significant digit is at the head of the string. An integer 25, for example, is stored in a string s as '2' at s[0], '5' at s[1], and null char at s[2].

YourSRN.c

YourSRN.c is your implementation file, which has the definition of all the functions declared in the header file ***intal.h***. If your SRN is PES1201700000 then the filename should be ***PES1201700000.c***. You can use the ***intal_sampletest.c*** for the sanity check. Compile ***intal_sampletest.c*** with ***YourSRN.c*** to sanity-check your implementation. The following commands may give a clearer picture for you.

```
prompt> ls
intal.h          intal_sampletest.c    YourSRN.c

prompt> gcc -Wall YourSRN.c intal_sampletest.c -o mytest.out

prompt> ./mytest.out
```

After you submit ***YourSRN.c***, we are going to assess by running with a hidden test file. That is going to have an exhaustive set of test cases, some may fail even though you could not catch it with the sample tests. Feel free to modify the given ***intal_sampletest.c*** and play with it to make sure your implementation is robust enough. Do not modify ***intal.h*** at any point in time because that is the interface/contract between you and us!

All the functions, whenever they return an intal, it should have stripped off the leading zeros. For example, the difference of "98" and "103" should be returned as "5", not as "05" or "005". However, the integer value of zero is represented as “0”. That is, return “0” for the difference of “35” and “35”. A parameter intal is never invalid or null and has at least a digit and a null termination. Also, intal as a parameter is not going to have any leading zeros. However, the integer value of zero is represented as “0”. That is, input to your functions is guaranteed to have a valid intal and also is guaranteed to never cross 1000 decimal digits for the return value. Function intal_bincoeff, for example, is never expected to have an answer having more than 1000 decimal digits. However, you need to make sure the intermediate values in your functions do not cross 1000 digits.

Whenever you are returning an intal, make sure you have allocated memory using **malloc(), calloc() or realloc()**. The test function is going to call **free()** on the returned intal immediately after the validation. It is guaranteed even the returning intal value is going to be less than 10^1000. Any other memory allocated by you should be freed by you before returning to the test function. Never return a parameter as the return value for an intal. Function intal_pow, for example, needs to allocate memory for the return value even in cases like a^1 where the answer is identical to a. No global variables should be used.

Suppose in `intal_gcd(intal_add(const char* intal1, const char* intal2)`, instead of writing:

```
if(strcmp(intal2, "0") == 0) return intal1;
```

write the following:

```
if(strcmp(intal2, "0") == 0) {
  char *temp = malloc((strlen(intal1) + 1) * sizeof(char));
  strcpy(temp, intal1);
  return temp;
}
```

Otherwise, the test function frees the returned value assuming it was allocated by the "add" function. That may result in failing some of the following tests because it has freed an intal that is used by the following tests.

Also, instead of writing:

```
b = intal_mod(a, b)
```

write the following:

```
char *temp = intal_mod(a, b);
char *temp2 = b;
b = temp;
free(temp2);
```

Otherwise, there is a memory leak and your tests may fail just because you have consumed more than the expected memory.

You are allowed to use the library functions from the following header files.
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

Do not include and use any other third-party library functions. Not even ***math.h*** functions. The library functions that are very close to the functions of **intal** are NOT allowed to be used. So, do NOT use **qsort()** and **bsearch()** declared in stdlib.h. Other than the definitions of the functions declared in the header file, you can have your own helper functions. Make sure to keep the helper functions by making them "static". I hope you know when we need static functions in C (https://www.tutorialspoint.com/static-functions-in-c). That way your helper functions won't name conflict with any functions used by the test functions we are going to use for the assessment.

YourSRN.txt

Submit a report ***YourSRN.txt*** of the mini-project as a simple ASCII text file. If your SRN is PES1201700000 then the filename should be ***PES1201700000.txt***. The report should contain the approach in implementing all the functions. Try to keep it concise yet covering all the details. Do not include the source code, which is already available in the implementation file. The report should have the following sections.

1. Introduction
    a. The section should answer questions like

        i. What is an intal?
        ii. How is it different from an integer in general and integer data types supported by C?
        iii. Applications of intal.
2. Approach
    a. Your approach in implementing the functionalities of intal.
3. Future work (If you had time and interest)
    a. Any more functionalities you think that can be included in the intal library?
    b. If you had no restrictions, how would you implement a library to handle integers of arbitrary length?

Submission

Deliverables

1. ***YourSRN.c***: Your implementation file, which has the definition of all the functions declared in the header file ***intal.h***. If your SRN is PES1201700000 then the filename should be ***PES1201700000.c***.
2. ***YourSRN.txt***: The report of your implementation. Write the approach in implementing all the functions. Try to keep it concise yet covering all the details. Do not include the source code, which is already available in the implementation file.

Submission

- Submit at the Google Form: https://forms.gle/HFxYw49DGdo8nK6r7 (Submission of the APSSSE Mini-project)
- Due date: 23:59 PM on **21th of May** 2020

FAQs

1. Is there any restriction on the use of macros for example **`#define MAX`** (for finding the max of two elements)?
    a. There are no such restrictions. Make sure you write your own macros, don't use third-party code.
2. Can we use **`typedef`** in our program at a global level?
    a. I don't think there would be any issues. At the moment, I'm not entirely sure of any name conflicts with the tests.
    a. It is alright to have struct definitions even though there is a slight chance of conflict with a similar struct in the test file. We are going to avoid having struct definitions in the test file for this purpose.
3. Can we use int64_t, int32_t and other fixed sized types, defined in the <inttypes.h> and <stdint.h> library of C99, for our project to get fixed-sized integers?
    a. As you are not allowed to the mentioned header files, you don't get to use those data types. My suggestion is, avoid C99 and stick to basic functionalities of ANSI C level (aka C90).

4. Do we have any time/space constraints other than those specified in the comments of the intal.h file?
    a. Some hints are given in the intal.h itself, which is mostly at the asymptotic complexity level. Memory leaks are discouraged anyway. I don't have a well-defined limitation in terms of absolute time and space limitations at the moment. Try to implement a reasonably good algorithm for the problem without bugs.
5. Is it safe to assume in intal_diff function that intal1 >= intal2?
    a. No. The difference is always a nonnegative integer. It is essentially the absolute_value_of(intal1 - intal2).
6. For all the functions can we assume that the inputs have no preceding '0's or should we right strip the inputs as well?
    a. There won't be any leading zeros in the arguments passed by the test functions. The integer zero is, however, will be "0".
7. Will **`intal_multiply()`** accept an O(n^2) solution or it has to be the Karatsuba algorithm?
    a. An O(n^2) solution is acceptable. I feel the Karatsuba algorithm is too much to ask for! Let me know if you have implemented the Karatsuba algorithm.
8. Could we get more edge cases in **`intal_sampletestcase.c`** to make sure our implementation guarantees to pass all the hidden test cases?
    a. No, sample tests are meant for understanding the problem along with a sanity check in terms of checking for compilation errors and making at least one call to each function. I'm not planning to cover all the edge cases which are going to be tested later in the assessment. You should play with the sample test file to test your code for the edge cases.
9. Can we use any library functions declared in the allowed header files like strtol(), qsort() and bsearch() declared in the stdlib.h?
    a. The library functions that are very close to the functions of **intal** are NOT allowed to be used. So, do NOT use **qsort()** and **bsearch()** declared in stdlib.h.
    b. As qsort() and bsearch() are too nearer to the problem you are solving, avoid using them. However, strtol() is not much of an use for you as it does not handle integers of arbitrary length.
10. Coin Row Problem: It is stated in the question that it should be implemented using Dynamic Programming. But it also mentions an O(1) space constraint. Is there a way to implement DP with O(1) space and is it necessary to?
    a. An intal itself takes a variable space (upto 1000 chars). DP method that uses just a constant width window of the DP table needs only O(1) intals to be stored at any point of time. If you keep the whole DP table, it consumes an extra space of O(n) intals.
11. Do we need to consider the case where the final answer exceeds a 1000 digits? If so, what are we supposed to do?
    a. You don't have to handle such cases. It is guaranteed that the inputs never expect an answer crossing the 1000 digit limit.
12. Is it possible to design an algorithm for the modulus operation in O(log intal1) time? Does it not also depend on intal2? If we created an algorithm to perform long division the way we did in high school and then get the remainder, would that suffice?
    a. Let me compare the O(log intal1) method with the O(intal1 / intal2) method. The O(intal1 / intal2) method works this way. Let r be intal1. Keep subtracting intal2 from r until r < intal2. The r you get is the answer. The value of r starts from intal1 and in each iteration r reduces by intal2. Therefore, the method takes O(intal1 / intal2) number of intal subtractions. This method will exceed the time limit in cases like the one in a sample test case. I was hinting on performing only O(log intal1) number of intal subtractions.

    b. The long division method (high school method) may work, but it mostly depends on how you manage intal subtractions there.

13. For the sorting function, to improve the performance can we only swap the pointers addresses and not copy the strings from one position to another?
    a. You are expected to swap or move only the pointers, not copying the char strings around.
14. I have made sure `intal_bincoeff(1000, 900)` takes much less time than `intal_bincoeff(1000, 500)`. But, `intal_bincoeff(1000, 500)` still takes more than one second. Does it exceed the time limit?
    a. `intal_bincoeff(1000, 500)` could exceed the time limit if we give that as a test. We are not going to give tests that take more time for an efficient implementation. There will be tests like `intal_bincoeff(1000, 900)` that are expected to run fast in an efficient implementation. If your implementation takes more time for that, we consider that as the time limit exceeded (TLE).
15. Is it alright to modify the array of intals as a parameter?
    a. intal_sort() expects you to sort them in-place. Other than intal_sort(), no other function expects you to modify the array. Treat it as immutable.
16. What does it mean by a O(log intal1) time taking algorithm for the **`intal_mod()`**?
    a. I have heard of a method in which you keep subtracting intal2 from intal1 until intal1 is less than intal2. The resulting intal1 is the expected answer. The number of subtractions it takes is intal1/intal2. That is what I meant by O(intal1/intal2) number of intal subtractions, which may exceed the time limit (Eg: intal1 = "1234512345123451234512345" and intal2 = “12”). If you can reduce to O(log intal1) number of intal subtractions, it is most likely going to be within the time limit.
17.