CMPUT275—Assignment 2 (Winter 2026)

R. Hackman

Due Date: Friday February 27th, 8:00PM

Per course policy you are allowed to engage in *reasonable* collaboration with your classmates. You must include in the comments of your assignment solutions a list of any students you collaborated with on that particular question.

In this assignment and all following assignments you should test against the sample executables. The sample executables try to help you out and print error messages when receiving invalid input (though they do not catch *all* invalid input). You do not have to replicate any error messages printed, unless the assignment specification specifically asks for error messages. These messages are only in the sample executable to help you catch when you write an invalid test case.

Memory Management: In order to complete some of these questions you will be required to use dynamic memory allocation. Your programs must not leak any memory, if you leak memory on a test case then you are considerd to have failed that test case. You can test your program for memory leaks by using the tool `valgrind`.

Memory Requirements: In addition to not leaking memory your programs must not use at any one time more than double of the maximum amount of memory they require. That is if implementing a dynamic array you may (and should) use the doubling strategy. If you simply allocate a very large array hoping input sizes will never exceed that then you will not receive marks for that question. For initializing dynamic arrays you may intialize them to have a capacity of 4.

Compilation Flags: each of your programs should be compiled with the following command:

```
gcc -Wall -Wvla -Werror
```

These are the flags we'll compile your program with, and should they result in a compilation error then your code will not be compiled and ran for testing.

Allowed libraries: `stdlib.h`, and `stdio.h`. No other libraries are allowed.

1. **Base-N Hamming Distance**

   The *Hamming Distance* between two equal length sequences is the number of positions in which they differ. For example the Hamming distance between two sequences "ABCD" and "AXCY" is two, because they differ in their second and fourth positions.

   Since numbers can be thought of as sequences of digits the hamming distance between two numbers can be computed. However, not all numbers are the same length, for this reason we will extend the definition Hamming distance for sequences of different lengths. When comparing two sequences A and B then their extended Hamming distance is the number of positions in which they differ, in the case that one sequence has an item at a position that the other sequence does not then they are considered to differ in that position (since the absence of a value is a difference from any possible value). For example, the base-10 numbers 10234

and 234 have an extended Hamming distance of 2, since they differ in the thousands and ten-thousandths position.

Two numbers Hamming distances are not an innate property. Rather, the Hamming distance of numbers is dependent on their base! For example the two numbers 65 and 97 have a Hamming distance of 2 in Base-10, however in base-2 these numbers are represented by the values 10000001 and 11000001 respectively, which differs in only one position.

Your goal for this question is to write `hamming.c` which defines a C program that compares two natural numbers in a particular base and prints out the base-10 number which represents the Hamming distance of those two numbers in the particular base (followed by a newline). Your program should take three command line arguments, the first represents one base-10 natural number, the second represents another base-10 natural number, and the third is another base-10 natural number greater than one which represents the base to compare the previous two numbers in. If your program is compiled to the executable named `hamming` in the current working directory then the following are example executions of it and the resultant output.

Example one command execution

```
./hamming 234 182 4
```

Example one output

```
3
```

Example two command execution

```
./hamming 234 182 2
```

Example two output

```
5
```

Example three command execution

```
./hamming 23452 18472 10
```

Example three output

```
3
```

You are guaranteed the integers you are given will never be larger than the largest unsigned integer representable in 4-bytes. That is, none of the arguments will be larger than 4294967295.

Note: While your program is comparing two numbers in a given base, the integers that are given as arguments are always written in base-10.

Deliverables For this question include in your final submission zip your c source code file named `hamming.c`

2. **Easy Strings**

As we have learned in C we do not have a complex data type representing strings built in. Instead, all we have are arrays of characters that include a null-terminator, often called C strings. While C does have standard functions defined for working with C strings these functions are simple and do not dynamically grow strings as necessary when more space is needed. We would like to be able to work with strings simply as we did in Python.

As such, we will define dynamically growing arrays of characters to represent strings as we have done with other dynamically growing arrays. For ease in differentiating between our string data type, the basic C string data type, and the abstract concept of a string of characters we will, in this specification, refer to our data type as an `SString`, the C data type as a C String (or C Str), and the abstract concept as simply a string.

Your job is to write a program that can handle a number of manipulations to four `SStrings` which will be referred to with the names `a`, `b`, `c`, and `d`. An `SString` is not represented by a null-terminated array of characters. Instead, an `SString` is represented by a pointer to an array of characters, an unsigned integer which represents the length of the string (how many characters are actually in the array), and a capacity of the array (which represents how many characters total the array can hold). Since a length of the `SString` is required to be maintained there is no need to include a null-terminator at the end of the characters — this means the arrays that comprise part of our `SStrings` are not valid C strings and would not work with the built-in string functions.

When initialized your program should construct all four `SStrings` as empty with a capacity of four. Throughout the run of your program you will have to manipulate these `SStrings` through various procedures defined below. Your `SStrings` must follow the *doubling strategy* discussed in class for allocating more memory when necessary. That is, when an `SString` requires more memory the capacity of it should be doubled.

Your program will need to read input that specifies several commands the user would like to do to interact with the four `SStrings` your program manages (which the user will refer to as `a`, `b`, `c`, and `d`).

Your program should read commands from the user executing them until either receiving EOF or the command `q` upon which time your program terminates. In each of the following commands `<targ>` must be replaced with one of `a`, `b`, `c`, or `d` to refer to that particular `SString`, and `<str>` must be replaced with a string. The commands your program must handle are:

- `r <targ> <str>` — the command for **r**eading in a string from standard input into the given `SString`. When you receive this command your program will read the given input string until it receives any whitespace character (or EOF). After reading this string in the target `SString` should represent that string exactly. If the given `SString` already had enough capacity to store the read-in string then the capacity will not have changed. If the given `SString` did not already have enough capacity to store the read-in string then it will have doubled as many times as necessary to store that string.
- `r <targ> "<str>"` — the same as the command above except instead of stopping reading the input string when seeing a whitespace the end of the string is denoted by the closing double quote. After reading this string in the target `SString` will represent exactly the string enclosed within the double quotes, following the same capacity rules as above.

- `p <targ>` — **p**rints out the string that is represented by the given `SString`, followed by a newline.
- `d <targ>` — displays the **d**etails of the given `SString`. Prints out the string it represents enclosed in quotes as well as the size and capacity values that are stored for that `SString`. As always the format is specified by the sample executable.
- `a <targ> <str>` — the command for **a**ppending a string from standard input to the given `SString`. When you receive this command your program will read the given input string until it receives any whitespace character (or EOF). After reading this string in the target `SString` should represent the result of appending the read in string to the current value of the `SString`.
- `a <targ> "<str>"` — the same as the command above except instead of stopping reading the input string when seeing a whitespace the end of the string is denoted by the closing double quote.
- `c <targ1> <targ2> <targ3>` — the command for **c**oncatenating two `SStrings`. This command implements the behaviour represented by `<targ1> = <targ2> + <targ3>` in Python, where each of the targets are Python strings. After this command is executed the `SString` referred to by `<targ1>` will be the result of concatenating `<targ2>` and `<targ3>`.
- `q` — the command for quitting your program, when received your program should terminate.

Note 1: For each command there may be *any* amount of whitespace in-between the components of the commands, or in-between separate commands. You must be able to handle this arbitrary amount of whitespace.

Note 2: Your `SStrings` must start at capacity 4 and double their capacity each time they grow. You are *never* to shrink your capacity of your `SStrings`. So for example, if an `SString` already has capacity of 32, but you execute a command that would result in that `SString` representing a string of only three characters it does not matter the capacity will still be 32.

Hint: With proper functional abstraction implementing the variety of functionality requested above is quite easy (in fact, once you have completed command `r`, `p`, and `d` you should have all the functionality you need to implement the other behaviours). At a bare minimum it is suggested you write a function to append a single character to an `SString`. That function should do all the necessary steps to append that character to the given `SString` (such as updated capacity and size, as well as the array of characters itself). That helper function alone will make the work of this question considerably easier.

Deliverables For this question include in your final submission zip your c source code file named `sstring.c`

3. **Image Translation**

In this question you will be developing a program that can apply some translations to image files. This question will operate on text that constitutes a "Plain or Raw PPM" image format. This image format is a plaintext format that is readable by a human. Each Plain PPM file begins with a header that gives you some information about the file, and then followed by the "payload" which is the image itself. The format of a Plain PPM image file is as follows:

- The first line will have the string `P3` indicating that this is a P3 PPM file.
- The second line contains two integers separated by whitespace, the first is the width of the image (in pixels) and the second is the height of the image (in pixels).
- The third line indicates the maximal value for any colour component of an individual pixel, for this assignment we will assume this is always 255.
- After the third line is the payload. There will be one line for each pixel in the height of your image.
- In each line in the payload there will be sets of three integers that represent the colour of an individual pixel. Each of these three integers in order will represent the red (R), green (G), and blue (B) value of that pixel. There will be a number of pixels described in each line equal to the width of your image.

For example, consider the following PPM file which has one row of blue pixels, below that a row of red pixels, and below that row of green pixels.

```
P3
4 3
255
0 0 255 0 0 255 0 0 255 0 0 255
255 0 0 255 0 0 255 0 0 255 0 0
0 255 0 0 255 0 0 255 0 0 255 0
```

Pixels are stored in order from the top left of the image to the bottom right. So the first pixel described in your PPM file is the leftmost pixel of the top row, the second pixel in that same line is the next pixel in the top row, so on and so forth. Each line represents one rows of pixels in your image.

You must write a program `transformer.c` that reads from standard input an image in the PPM format. Your program must also accept optionally the command line arguments `-f` and `-s`. Depending on which (if any) command line arguments your program received, your program will then apply transformations to the image it read in. Your program must then print to standard output the transformed image (or the original image, if no transformations were requested). The two command line arguments have the following behaviour:

- `-f` indicates that the user of your program wants to flip the image. Flipping the image means that in each row the pixel at index 0 becomes the pixel at index n-1 (and vice versa), the pixel at index 1 becomes the pixel at index n-2 (and vice versa), etc.
- `-s` indicates the user of your program wants to apply the sepia filter to the image. For our purposes the sepia filter will exactly be determined by applying the following transformation to every pixels R, G, and B values:

$$newR = min(255, \lfloor R * 0.393 + G * 0.769 + B * 0.189 \rfloor)$$
$$newG = min(255, \lfloor R * 0.349 + G * 0.686 + B * 0.168 \rfloor)$$
$$newB = min(255, \lfloor R * 0.272 + G * 0.534 + B * 0.131 \rfloor)$$

Hint 1: It will be very useful to use a `struct` to represent a pixel! A simple struct containing 3 `ints` will work very well!

Hint 2: Once you know how you are going to store your image, this program can be broken down very nicely into a few functions. It is worth writing one function each to do the following operations: read the image in, apply a sepia filter to an image, flip an image, and print an image out.

Deliverables: For this question include in your final submission zip your c source code file named `transformer.c`

How to submit: Create a zip file `a2.zip`, make sure that zip file contains your C source code files `int_set.c`, `rpn.c`, and `transformer.c`. Assuming all three of these files are in your current working directory you can create your zip file with the command

```
$ zip a2.zip rpn.c int_set.c transformer.c
```

Upload your file `a2.zip` to the a1 submission link on eClass.