Programming Assignment 2 (PA2) - myCipher

Milestone Due: **<u>Wednesday, April 27 @ 11:59pm</u>**
Final Due: **<u>Tuesday, May 3 @ 11:59 pm</u>**

Assignment Overview

Hackers abound and you have been asked to write a crypto program to encrypt/decrypt data. The program will be able to read the data to be encrypted from `stdin` (data could either be typed in at the keyboard or `stdin` redirected from a file or via a pipe). The encrypted data will be written to `stdout` which can be redirected to a file.

You will be writing a program that takes 4 inputs from the command line:

```
[cs30xzzz@ieng9]:pa2$ ./pa2 passphrase key0 key1 rotateValue
```

The program will ask the user to enter a passphrase of at least 8 characters, two 32-bit crypto keys, and a rotation key in the range [-63, +63]. The program will XOR the first 8 characters of the passphrase and the two 32-bit crypto keys to form a 64-bit crypto mask. This mask will be used to XOR the data in 8 byte chunks (two 32-bit register operations per 8 byte chunk) plus individual single byte masks for the trailing bytes. With each 8 bytes of data encrypted with the 64-bit mask, the mask will be rotated according to the rotation key value (rotating left if the rotation key is negative and rotating right if the rotation key is positive).

Grading

- **README: 10 points** - See README File section
- **Compiling: 5 points** - Using our Makefile; no warnings. If what you turn in does not compile with the given Makefile, you will receive 0 points for this assignment. **NO EXCEPTIONS!**
- **Style: 10 points** - See Style Requirements section
- **Correctness: 75 points**
  - **Milestone (15 points)** - To be distributed across the Milestone functions (see below)
  - Make sure you have all files tracked in Git.
- **Extra Credit: 5 points** - View Extra Credit section for more information.
- **Wrong Language:** You will lose 10 points for each module in the wrong language, C vs. Assembly or vice versa.

NOTE: If what you turn in does not compile with given Makefile, you will receive 0 points for this assignment.

Getting Started

Follow these steps to acquire the starter files and prepare your Git repository.

<u>Gathering Starter Files:</u>
The first step is to gather all the appropriate files for this assignment.
Connect to ieng9 via ssh (replace cs30xzzz with YOUR cs30 account).

```
$ ssh cs30xzzz@ieng9.ucsd.edu
```

Create and enter the pa2 working directory.

```
$ mkdir ~/pa2
$ cd ~/pa2
```

Copy the starter files from the public directory.

```
$ cp -r ~/../public/pa2StarterFiles/* ~/pa2/
```

Copy your `isInRange.s` from your `pa1` directory.

```
$ cp ~/pa1/isInRange.s ~/pa2/
```

Starter Files Provided:		
`pa2.h`	`pa2Strings.h`	`pa2Globals.c`
`test.h`	`testrotate.c`	`Makefile`

Preparing Git Repository:
Refer to previous writeups for preparing your Git repository. This will be required again for PA2.

Sample Output

A sample stripped executable provided for you to try and compare your output against is available in the public directory. Note that you cannot copy it to your own directory; you can only run it using the following command (where you will also pass in the command line arguments):

```
$ ~/../public/pa2test
```

If there is a discrepancy between the sample output in this document and the `pa2test` output, follow the `pa2test` output.

Below are some brief example outputs of this program. Make sure you experiment with the public executable to further understand the program behavior. Bolded text is what you type in the terminal.

1. Command-line Parsing Errors

1.1. No arguments.

```
[cs30xzzz@ieng9]:pa2$ ./pa2

Usage: ./pa2 passphrase key0 key1 rotateValue
    passphrase   (must be at least eight characters long)
    key0         (must be numeric; decimal, octal, or hexadecimal)
    key1         (must be numeric; decimal, octal, or hexadecimal)
    rotateValue  (must be a decimal value within the range [-63 - +63])
```

1.2. Too many arguments (extra operand).

```
[cs30xzzz@ieng9]:pa2$ ./pa2 cs301234 45 20 15 23

Usage: ./pa2 passphrase key0 key1 rotateValue
    passphrase   (must be at least eight characters long)
    key0         (must be numeric; decimal, octal, or hexadecimal)
    key1         (must be numeric; decimal, octal, or hexadecimal)
    rotateValue  (must be a decimal value within the range [-63 - +63])
```

2. Other Errors

2.1. Passphrase is too short and rotate value is outside the required range.

```
[cs30xzzz@ieng9]:pa2$ ./pa2 cse30 12 55 81

        Passphrase must be at least 8 chars long

        Rotation value must be within the range of [-63 - +63]

        --- Found 2 error(s) ---
```

2.2. `key0` is too large to be converted to an integer, and `key1` and the rotate value have invalid characters.

```
[cs30xzzz@ieng9]:pa2$ ./pa2 cse30rules 9999999999999999999999999 55a 0x12

        Converting "9999999999999999999999999" base "0": Result too large

        "55a" is not an integer

        "0x12" is not an integer

        --- Found 3 error(s) ---
```

3. Valid Output

3.1. Reading directly from `stdin`, writing directly to `stdout`.

```
[cs30xzzz@ieng9]:pa2$ ./pa2 cse30rocks 12 12 4
this is my first message
^D�__ �nU�SVB�RE �`Ce
```

3.2. Encrypting data: piping an input file to `stdin` and redirecting the output to an output file.

```
[cs30xzzz@ieng9]:pa2$ cat data | ./pa2 cse30pebbles 0xAB 056 -19 > dataCrypt
```

3.3. Decrypting data: redirecting an input file to `stdin` and redirecting the output to an output file.

```
[cs30xzzz@ieng9]:pa2$ ./pa2 cse30boulders 071 0xEE 63 < dataCrypt > dataDecrypt
```

Detailed Overview

The function prototypes for the various C and Assembly functions are as follows.

C routines:

```
long parsePassphrase( char * str, unsigned char * passphrase );
long parseKey( char * str, unsigned long * key );
int main( int argc, char * argv[] );
```

Assembly routines:

```
void createMask( unsigned long keys[], unsigned char passphrase[],
                 unsigned long mask[] );
int isInRange( long minRange, long maxRange, long value, long exclusive );
void myCipher( FILE * inFile, unsigned long mask[], long rotateValue );
long parseRotateValue( char * str, long * rotateValue );
void rotate( unsigned long mask[], long rotateValue );
```

For the Milestone, you will need to complete:

`createMask.s` `rotate.s` `parseKey.c` `parseRotateValue.s`

Process Overview:

The following is an explanation of the memory and logical components of the main tasks of the assignment, broken into 3 parts.

1. Parse command line arguments in pa2.c

There are 4 expected user inputs: `passphrase`, `key0`, `key1`, and `rotateValue`. Within `main()` in `pa2.c`, you will be parsing the command line arguments, checking for errors. The `passphrase`, `key0`, and `key1` will be used to create a 64-bit crypto mask. This mask will be used to encrypt/decrypt the user's data passed in through `stdin`.

You will be utilizing `parsePassphrase()`, `parseKey()`, and `parseRotateValue()` to process the command line arguments.

a) Check for errors (detailed in the file description for `pa1.c`).

b) If any errors are detected, print the appropriate error messages as the errors are found. Once all error conditions have been checked, report the number of errors if any were detected and return `EXIT_FAILURE`. Otherwise, if no errors were found, continue to step 2.

2. Create the mask

Create the mask by XORing the passphrase with the keys (see the file description for `createMask.s`).

3. Encrypt/decrypt data

You are now ready to encrypt/decrypt the data entered by the user through `stdin` by passing the appropriate arguments to `myCipher()`. Note that the act of encrypting and decrypting are the same in this case. That is, if you run the encryption on a file, you will get encrypted data. Then if you run the encryption on that encrypted data, you will get the original file back.

The main idea of `myCipher()` is to:

(1) Read the user's data from `stdin`
(2) Encrypt the data
(3) Write the encrypted data to `stdout`.

If your `myCipher()` implementation is working properly, you should be able to:

(1) Create an input file (let's say it's named `input`).
(2) Encrypt the file by piping `input` to your `pa2` executable and redirecting the encrypted data to an output file (let's say it's named `inputEncrypt`).
(3) Decrypt the file by piping `inputEncrypt` to the public `pa2test` executable and redirecting the decrypted data to an output file (let's say it's named `inputDecrypt`).
(4) Then if you run `diff` on the two files (`input` and `inputDecrypt`), no differences should be found.

C Functions to be Written

Listed below are the modules to be written in C.

---

parsePassphrase.c

`long parsePassphrase( char * str, unsigned char * passphrase );`

Parse the passphrase from the command line arguments by checking if `str` contains at least `PASSPHRASE_SIZE` characters. If it is at least this long, copy just the first `PASSPHRASE_SIZE` characters into the `passphrase` output parameter.

IMPORTANT:

- Do not use `strncpy()` to copy the passphrase because `passphrase` is an `unsigned char *`, and `strncpy()` expects a (signed) `char *` (see `man strncpy`). Therefore you must individually copy over the first `PASSPHRASE_SIZE` characters one at a time.
- Do not think of `passphrase` as a string; `passphrase` is just an array of 8 bytes of hex values and is NOT null terminated.

<u>Reasons for error:</u>

- If the `str` is shorter than the minimum `PASSPHRASE_SIZE`, return `LENGTH_ERR`.

<u>Return Value</u>: If errors were encountered, return the appropriate error value as indicated in the reasons for error section. Otherwise, the passphrase is stored in the output parameter (`passphrase`), and 0 is returned on success.

---

parseKey.c

`long parseKey( char * str, unsigned long * key );`

This module will be used to parse the second and third command line arguments, `key0` and `key1`. Parse the key passed in as `str` by converting the string to an unsigned long. The user can enter these values in decimal, octal, or hexadecimal.

<u>Things to consider:</u>

- How can you convert a string to an unsigned long where the value can be expressed in either decimal, octal, or hexadecimal? (hint: `man -s3c strtoul()`)
- How can you check if errors occurred during the conversion?

<u>Reasons for error:</u>

- If the number was too large to be successfully converted, return `ERANGE_ERR`.
- If the number contained invalid characters, return `ENDPTR_ERR`.

<u>Return Value</u>: If the conversion was successful, the key is stored in the output parameter (`key`), and 0 is returned. Otherwise, return the appropriate error value as indicated in the reasons for error section.

pa2.c

```
int main( int argc, char * argv[] );
```

This function is the main driver for the program. It will first parse all of the command line arguments. If all arguments are valid, it will create the 64-bit crypto mask and perform the encryption/decryption from `stdin`. Otherwise, the appropriate error messages will be printed.

IMPORTANT: You must include the following line at the beginning of your `main()` function. It will disable buffering on `stdout` which will help in matching the output of the test program. This line must be included to receive full credit on all tests.

```
(void) setvbuf( stdout, NULL, _IONBF, 0 );
```

Parsing command line arguments:

1. Check if the correct number of command line arguments were passed in. If there are an invalid number of arguments, print the usage string and return `EXIT_FAILURE`.
2. First parse the `passphrase`. You will need to initialize an array of `PASSPHRASE_SIZE` unsigned characters. Initialize each byte of the `passphrase` with an easily recognizable hexadecimal value such as `0xA5` (defined as `INIT_PASSPHRASE` in `pa2.h`). This will make debugging easier by being able to quickly identify if the `passphrase` has been correctly set by `parsePassphrase()`. If `passphrase` is still `0xA5A5A5A5A5A5A5A5` after calling `parsePassphrase()`, then you know there is a problem with your parsing routine. If `parsePassphrase()` returned an error, print the appropriate error message.
3. Parse the rest of the command line arguments by calling their respective parsing modules. After parsing each argument, if an error was indicated, print the appropriate error message and continue parsing the remaining arguments. Remember, for any error where `errno` was set, use `snprintf()` to construct the error string, and then `perror()` to print out the complete error message. (All error strings are located in `pa2Strings.h`.)
4. If any errors occurred, print the number of errors encountered and return `EXIT_FAILURE`.

If no errors were encountered, perform the encryption/encryption:

1. Create the 64-bit crypto mask from the `passphrase`, `key0`, and `key1`.
2. Utilize the `myCipher()` method to encrypt the data from `stdin`, using the `mask` and the `rotateValue`.

Reasons for error:

- Incorrect number of command line arguments are passed in
- `passphrase` does not meet the minimum length requirement
- `key0`, `key1`, or `rotateValue` are too large to be converted to longs
- `key0`, `key1`, or `rotateValue` contain invalid characters and cannot be converted to longs
- `rotateValue` is not within the valid range

Return Value: If errors were encountered, return `EXIT_FAILURE`. Otherwise, return `EXIT_SUCCESS`.

Assembly Functions to be Written

Listed below are the modules to be written in Assembly.

---

createMask.s

```
void createMask( unsigned long keys[], unsigned char passphrase[],
                 unsigned long mask[] );
```

This module creates the 64-bit crypto mask that will later be used to encrypt the data. The 64-bit mask will be stored in `mask` as an array of two 32-bit mask values. Create the mask by XORing the passphrase with the keys. This will require loading the appropriate values from `keys` and `passphrase`, and storing the results in `mask`.

More succinctly, this module should perform the following:

```
mask[0] = keys[0] ^ (1st half of passphrase)
mask[1] = keys[1] ^ (2nd half of passphrase)
```

Return Value: None. Store the 64-bit crypto mask in the output parameter `mask`.

---

isInRange.s

```
int isInRange( long minRange, long maxRange, long value, long exclusive );
```

Copied from PA1, no changes necessary.

---

myCipher.s

```
void myCipher( FILE * inFile, unsigned long mask[], long rotateValue );
```

This function is responsible for the encryption/decryption of the user input using the 64-bit crypto `mask` created from the command line arguments. You will be reading in the user input from the `inFile` in blocks of `BUFSIZ` bytes. From each block read, you will encrypt the data 8 bytes at a time. To do this, first rotate the mask by `rotateValue`, and then XOR the 8-byte mask with each 8-byte chunk of the block of `BUFSIZ` bytes. If there are less than 8 bytes left in the block, each byte must be encrypted individually. Be sure to only rotate the mask by `rotateValue` a single time before handling these last bytes (do **NOT** rotate per individual byte). Once you have encrypted a block of up to `BUFSIZ` bytes of data, write the encrypted data to stdout.

Note: You will need to create an assembler constant for `BUFSIZ` = 1024.

Things to consider:

- How would you XOR 8 bytes of data with the mask represented by two unsigned longs?
- How would you encrypt individual bytes of data with individual bytes of the mask?
- How do you read data from a `FILE *`? (hint: `man -s3c fread`)
- How do you write data to a `FILE *`? (hint: `man -s3c fwrite`)
- What exactly is a `FILE *`? (hint: `man -s3c stdio`)

Return Value: None

parseRotateValue.s
`long parseRotateValue( char * str, long * rotateValue );`

This function will convert the rotate value passed in as a command line argument from a string to a long (interpreted as a decimal value), and will check if it is in the required range of [`MIN_ROTATE` - `MAX_ROTATE`], inclusive (make sure you use your `isInRange()` function). The parsed rotate value will be stored in the output parameter `rotateValue` and the return value will be used to indicate errors.

Make sure you use the global variables defined in `pa2Globals.c`. Remember, you need to load these values before using them in your assembly routine.

Things to consider:

- How can you convert a string to a long as a decimal value?
- How can you check if errors occurred during the conversion?

Reasons for error:

- If the number was too large to be successfully converted, return `ERANGE_ERR`.
- If the number contained invalid characters, return `ENDPTR_ERR`.
- If the number was outside the required range, return `BOUND_ERR`.

Return Value: If the conversion was successful, the rotate value is stored in the output parameter (`rotateValue`), and 0 is returned. Otherwise, return the appropriate error value as indicated in the reasons for error section.

---

rotate.s
`void rotate( unsigned long mask[], long rotateValue );`

This function will rotate the bits in the 64-bit crypto `mask`. The `rotateValue` indicates how many bits to rotate by. A negative `rotateValue` indicates the bits will be rotated left, and a positive `rotateValue` indicates the bits will be rotated right. You should perform the rotation one bit at a time. After performing the rotation, store the rotated mask back into the `mask` parameter.

For example: the following shows the result of rotating a 64-bit mask by -16 bits:

Before rotate: `0xCAFEBABEDEADBEEF`
After rotate: `0xBABEDEADBEEFCAFE`

Return Value: None. Store the rotated mask in the output parameter `mask`.

Unit Testing

You are provided with a basic unit test file for `rotate.s`. This has minimal test cases and is only meant to give you an idea of how to write your own tests.

You must write unit test files for each of the Milestone functions, as well as add several of your own thorough test cases to all 4 unit test files. You will lose points if you don't do this! You are responsible for making sure you thoroughly test your functions. Make sure you think about boundary cases, special cases, general cases, extreme limits, error cases, etc. as appropriate for each function.

The Makefile includes the rules for compiling and running your Milestone function tests. Keep in mind that your unit tests will not build until all required files for the unit tests have been written (see the Makefile for proper target names).

These test files are not being collected for the Milestone and will only be collected for the final turnin (however, they should already be written by the time you turn in the Milestone because you should be using them to test your Milestone functions).

Unit tests you need to complete:

`testcreateMask.c`
`testrotate.c`
`testparseKey.c`
`testparseRotateValue.c`

To compile:

```
$ make testrotate
```

To run:

```
$ ./testrotate
```

(Replace "`testrotate`" with the appropriate file names to compile and run the other unit tests)

README File

Your README file for this and all assignments should contain:

- High level description of what your program does.
- How to compile it (be more specific than: just typing "make"--i.e., what directory should you be in?, where should the source files be?, etc.).
- How to run it (give an example).
- An example of normal output and where that normal output goes (stdout or a file or ???).
- An example of abnormal/error output and where that error output goes (stderr usually).
- How you tested your program (what test values you used to test normal and error states) showing your tests covered all parts of your code (test coverage). (Be more specific than diff'ing your output with the solution output--i.e., what are some specific test cases you tried?, what different types of cases did you test?, etc.)
- Anything else that you would want/need to communicate with someone who has not read the assignment write-up but may want to compile and run your program.
- Answers to questions (if there are any).

Questions to Answer in the README

1. What is the command to rename a file?
2. What is the command to copy a file?
3. What happens when you select text and then middle click in the vim editor when in insert/input mode?
4. What is a .vimrc file, and how do you create/edit them?
5. What is the command to cut a full line of text to the clipboard in vim? How do you paste it? (Both the questions refer to using the keyboard, not using the mouse).
6. How do you search for a string in vim?
7. How do you turn on line numbers in vim?
8. How can you quickly (with a single Linux command) change directory to a directory named fubar that is in your home (login) directory? You cannot change directory to your home directory first and then

change directory to fubar. That would take two commands. State how you would do this with a single command no matter where your current working directory might be.

9. How do you change the permissions on a file? Let's say want to give read permission to the group? Specify the command to do this.
10. Why are professional engineers expected to act with integrity?

Extra Credit

There are 5 points total for extra credit on this assignment.

- Early turnin: **[2 Points]** 48 hours before regular due date and time
  [1 Point] 24 hours before regular due date and time
  (it's one or the other, not both)
- **[3 Points Total, 0.5 for each nop]** Eliminating nops in the sample assembly file.

Getting Started

Copy over the following files from the public directory.

```
$ cp ~/../public/isortDriver.c ~/pa2
$ cp ~/../public/isort.s ~/pa2
```

Overview

You will be modifying `isort.s` to perform assembly optimization. This program randomly populates an array of 400 ints, then calls the `isort()` method in order to perform an insertion sort. The `isort()` method takes in an array of ints (which, you know that the name of the array is actually a pointer to the first element of the array) and length of the array. After insertion sort has completed, the program calculates the value of the maximum integer value in the array minus the minimum integer value of the array.

There are a total of 6 nops in the assembly code (`isort.s`). Your task is to eliminate as many of the nops as you can. All nops can be eliminated in `isort.s`. Every nop eliminated will be worth half a point, so to get all 3 points you will have to eliminate all 6 nops. If the optimized version does not have the same output as the unoptimized version, no points will be awarded.

NOTE:

- Only `isort.s` should have assembly optimization for extra credit. Do not modify any other assembly functions for the PA2 assignment.
- Make sure you do not make any changes to `isortDriver.c`. All the optimization changes you need to make should be in `isort.s`.

Compiling

You can compile the extra credit program using the following command.

```
$ gcc -o isort isortDriver.c isort.s
```

Sample Output

```
[cs30xzzz@ieng9]:pa2$ ./isort
   20    24    26    29    44    99   122   148   159   199
  229   296   338   352   368   398   403   403   444   514
  560   565   587   598   607   613   643   653   659   686
  736   783   800   807   810   835   850   858   912   922
  944   966   973  1016  1047  1060  1154  1237  1241  1242
 1243  1249  1281  1392  1422  1456  1477  1481  1483  1511
 1514  1562  1606  1810  1886  1911  1915  1917  1934  1945
```

```
 1950  2020  2029  2036  2044  2052  2068  2089  2091  2132
 2133  2200  2218  2232  2240  2254  2270  2286  2300  2304
 2313  2346  2351  2367  2397  2440  2441  2446  2462  2504
 2553  2619  2624  2639  2647  2667  2671  2676  2760  2765
 2872  2905  2908  2944  2967  3004  3015  3027  3036  3078
 3144  3163  3212  3271  3275  3321  3346  3365  3371  3383
 3403  3428  3443  3449  3451  3451  3508  3508  3553  3580
 3632  3640  3644  3650  3733  3772  3783  3785  3786  3811
 3886  3898  3954  3972  3989  4015  4045  4045  4115  4116
 4119  4126  4130  4160  4182  4184  4207  4246  4324  4344
 4426  4430  4431  4444  4458  4461  4501  4612  4663  4721
 4728  4776  4807  4856  4873  4890  4908  4976  4978  5013
 5024  5060  5103  5158  5195  5261  5271  5297  5313  5323
 5335  5342  5363  5439  5536  5563  5596  5640  5679  5691
 5703  5711  5744  5788  5789  5826  5839  5847  5861  5868
 5885  5896  5928  5975  6000  6003  6084  6087  6096  6138
 6190  6300  6317  6329  6402  6402  6417  6425  6547  6562
 6587  6591  6624  6711  6719  6735  6746  6758  6782  6811
 6820  6821  6865  6865  6873  6876  6885  6902  6917  6934
 6960  7045  7076  7096  7119  7163  7174  7209  7257  7318
 7318  7339  7341  7373  7374  7418  7450  7497  7497  7500
 7500  7512  7527  7550  7567  7586  7590  7601  7610  7614
 7617  7623  7629  7682  7711  7724  7749  7762  7767  7797
 7829  7857  7902  7909  7978  7986  8069  8092  8094  8101
 8153  8211  8269  8418  8440  8453  8483  8487  8508  8545
 8573  8598  8628  8638  8650  8661  8710  8727  8751  8762
 8795  8816  8921  8997  8999  9041  9076  9095  9149  9173
 9188  9245  9261  9319  9375  9385  9421  9427  9449  9454
 9518  9519  9555  9646  9687  9708  9767  9788  9896 10022
10076 10138 10152 10168 10225 10229 10238 10246 10293 10304
10309 10334 10368 10433 10508 10565 10589 10662 10671 10752
10797 10874 10882 10885 10904 10908 10948 10984 11100 11155
11257 11327 11392 11419 11427 11499 11502 11565 11604 11647
11677 11684 11684 11687 11716 11762 11889 11920 12024 12045
12059 12063 12125 12137 12153 12234 12237 12245 12245 12291

Max - Min = 12271
```

Milestone Turn-in Instructions

Milestone Turn-in - due Wednesday night, April 27 @ 11:59 pm [15 points of Correctness Section]

Before final and complete turnin of your assignment, you are required to turnin several modules for the Milestone check.

Files required for the Milestone:			
`createMask.s`	`rotate.s`	`parseKey.c`	`parseRotateValue.s`

Each module must pass all of our unit tests in order to receive full credit.

A working Makefile with all the appropriate targets and any required header files must be turned in as well. All Makefile test cases for the milestone functions must compile successfully via the commands `make test***`.

In order for your files to be graded for the Milestone Check, you must use the milestone specific turnin script.

```
$ cse30_pa2milestone_turnin
```

To verify your turn-in:

```
$ cse30verify pa2milestone
```

Final Turn-in Instructions

Final Turn-in - due Tuesday night, May 3 @ 11:59 pm

Once you have checked your output, compiled, executed your code, and finished your README file (see above), you are ready to turn it in. Before you turn in your assignment, you should do `make clean` in order to remove all the object files, lint files, core dumps, and executables.

Files required for the Final Turn-in:

```
createMask.s          parseRotateValue.s    pa2Globals.c
isInRange.s           rotate.s              pa2Strings.h
myCipher.s            pa2.c                 Makefile
parseKey.c            pa2.h                 README
parsePassphrase.c

testcreateMask.c      testrotate.c          testparseKey.c
testparseRotateValue.c
```

Extra Credit Files:

```
isortDriver.c         isort.s
```

Use the above names *exactly* otherwise our Makefiles will not find your files.

How to Turn in an Assignment

Use the following turnin script to submit your full assignment before the due date as follows:

```
$ cse30turnin pa2
```

To verify your turn-in:

```
$ cse30verify pa2
```

Up until the due date, you can re-submit your assignment via the scripts above. Note, if you turned in the assignment early for extra credit and then turned it in again later (after the extra credit cutoff), you will no longer receive early turn-in credit.

Failure to follow the procedures outlined here will result in your assignment not being collected properly and will result in a loss of points. Late assignments WILL NOT be accepted.

If there is anything in these procedures which needs clarifying, please feel free to ask any tutor, the instructor, or post on the Piazza Discussion Board.

Style Requirements

You will be graded on style for all the programming assignments. The requirements are listed below. Read carefully, and if any of them need clarification do not hesitate to ask.

- Use reasonable comments to make your code clear and readable.
- Use file headers and function header blocks to describe the purpose of your programs and functions. Sample file/function headers are provided with PA0.
- Explicitly comment all the various registers that you use in your assembly code.
- In the assembly routines, you will have to give high level comments for the synthetic instructions, specifying what the instruction does.
- You should test your program to take care of invalid inputs like non-integers, strings, no inputs, etc. This is very important. Points will be taken off if your code doesn't handle exceptional cases of inputs.
- Use reasonable variable names.
- Error output goes to stderr. Normal output goes to stdout.
- Use #defines and assembly constants to make your code as general as possible.
- Use a local header file to hold common #defines, function prototypes, type definitions, etc., but not variable definitions.
- Judicious use of blank spaces around logical chunks of code makes your code easier to read and debug.
- Keep all lines less than 80 characters, split long lines if necessary.
- Use 2-4 spaces for each level of indenting in your C source code (do not use tab). Be consistent. Make sure all levels of indenting line up with the other lines at that level of indenting.
- Do use tabs in your Assembly source code.
- Always recompile and execute your program right before turning it in just in case you commented out some code by mistake.
- Do #include only the header files that you need and nothing more.
- Always macro guard your header files (#ifndef ... #endif).
- Never have hard-coded magic numbers (any number other than -1, 0, or 1 is a magic number). This means we shouldn't see magic constants sitting in your code. Use a #define if you must instead.