CSE 220: Systems Fundamentals I

Stony Brook University

Homework Assignment #3

Fall 2018

Assignment Due: November 9, 2018 by 11:59 pm

Important Information about CSE 220 Homework Assignments

- Read the entire homework documents twice before starting. Questions posted on Piazza whose answers are clearly stated in the documents will be given lowest priority by the course staff.
- When writing assembly code, try to stay consistent with your formatting and to comment as much as possible. It is much easier for your TAs and the professor to help you if we can quickly figure out what your code does.
- You personally must implement homework assignments in MIPS Assembly language by yourself. You may not write or use a code generator or other tools that write any MIPS code for you. You must manually write all MIPS Assembly code you submit as part of the assignments.
- Do not copy or share code. Your submissions will be checked against other submissions from this semester and from previous semesters.
- You must use the Stony Brook version of MARS posted on Piazza. Do not use the version of MARS posted on the official MARS website. The Stony Brook version has a reduced instruction set, added tools, and additional system calls you will need to complete the homework assignments.
- Do not submit a file with the functions/labels `main` or `_start` defined. You are also not permitted to start your label names with two underscores (`__`). You will obtain a zero for an assignment if you do this.
- Submit your final `.asm` file to Blackboard by the due date and time. Late work will not be accepted or graded. Code that crashes and cannot be graded will earn no credit. No changes to your submission will be permitted once the deadline has passed.

How Your CSE 220 Assignments Will Be Graded

With minor exceptions, all aspects of your homework submissions will be graded entirely through automated means. Grading scripts will execute your code with input values (e.g., command-line arguments, function arguments) and will check for expected results (e.g., print-outs, return values, etc.) For this homework assignments you will be writing *functions* in assembly language. The functions will be tested independently of each other. This is very important to note, as you must take care that no function you write ever has side-effects or requires that other functions be called before the function in question is called. Both of these are generally considered bad practice in programming.

Some other items you should be aware of:

- All test cases must execute in 100,000 instructions or fewer. Efficiency is an important aspect of programming. This maximum instruction count will be increased in cases where a complicated algorithm might be

necessary, or a large data structure must be traversed.

- Any excess output from your program (debugging notes, etc.) might impact grading. Do not leave erroneous print-outs in your code.
- We will provide you with a small set of test cases for each assignment to give you a sense of how your work will be graded. It is your responsibility to test your code thoroughly by creating your own test cases.
- The testing framework we use for grading your work will not be released, but the test cases and expected results used for testing will be released.

Getting Started

Visit Piazza and download the file `hw3.zip`. Decompress the file and then open `hw3.zip`. Fill in the following information at the top of `hw3.asm`:

1. your first and last name as they appear in Blackboard
2. your Net ID (e.g., jsmith)
3. your Stony Brook ID # (e.g., 111999999)

Having this information at the top of the file helps us locate your work. If you forget to include this information but don't remember until after the deadline has passed, don't worry about it – we will track down your submission.

Inside `hw3.asm` you will find several function stubs that consist simply of `jr $ra` instructions. Your job in this assignment is implement all the functions as specified below. Do not change the function names, as the grading scripts will be looking for functions of the given names. However, you may implement additional helper functions of your own, but they must be saved in `hw3.asm`. Helper functions will not be graded.

If you are having difficulty implementing these functions, write out pseudocode or implement the functions in a higher-level language first. Once you understand the algorithm and what steps to perform, then translate the logic to MIPS assembly code.

Be sure to initialize all of your values (e.g., registers) within your functions. Never assume registers or memory will hold any particular values (e.g., zero). MARS initializes all of the registers and bytes of main memory to zeroes. The grading scripts will fill the registers and/or main memory with random values before calling your functions.

Finally, do not define a `.data` section in your `hw3.asm` file. A submission that contains a `.data` section will probably receive a score of zero.

Register Conventions

You must follow the register conventions taught in lecture and reviewed in recitation. Failure to follow them will result in loss of credit when we grade your work. Here is a brief summary of the register conventions and how your use of them will impact grading:

- It is the callee's responsibility to save any `$s` registers it overwrites by saving copies of those registers on the stack and restoring them before returning.
- If a function calls a secondary function, the caller must save `$ra` before calling the callee. In addition, if the caller wants a particular `$a`, `$t` or `$v` register's value to be preserved across the secondary function

call, the best practice would be to place a copy of that register in an `$s` register before making the function call.

- A function which allocates stack space by adjusting `$sp` must restore `$sp` to its original value before returning.
- Registers `$fp` and `$gp` are treated as preserved registers for the purposes of this course. If a function modifies one or both, the function must restore them before returning to the caller. There is no reason for your code to touch the `$gp` register, so leave it alone.

The following practices will result in loss of credit:

- "Brute-force" saving of all `$s` registers in a function or otherwise saving `$s` registers that are not overwritten by a function.
- Callee-saving of `$a`, `$t` or `$v` registers as a means of "helping" the caller.

How to Test Your Functions

To test your implementated functions, open the provided `hw3_main` files in MARS. Next, assemble the `main` file and run it. MARS will include the contents of any `.asm` files referenced with the `.include` directive(s) at the end of the file and then add the contents of your `hw3.asm` file before assembling the program.

Each main file calls a single function with one of the sample test cases and prints any return value(s). You will need to change the arguments passed to the functions to test your functions with the other cases.

To test each of your functions thoroughly, write your own main programs (new individual files) and create your own function arguments in those main files. Your submission will not be graded using the examples provided in this document or the provided main file(s).

Any modifications to the `main` files will not be graded. You will submit only your `hw3.asm` for grading. Make sure that all code required for implementing your functions is included in the `hw3.asm` file. To make sure that your code is self-contained, try assembling your `hw3.asm` file by itself in MARS. If you get any errors (such as a missing label), this means that you need to refactor (reorganize) your code, possibly by moving labels you inadvertently defined in a `main` file (e.g., a helper function) to `hw3.asm`.

The ADFGVX Cipher

The primary objectives of this homework assignment are to learn how to index and iterate through two dimensional arrays of data and to reinforce proper register conventions in MIPS assembly language.

You will be implementing the historically significant ADFGVX encryption cipher. Before reading any more of this assignment, read the linked Wikipedia article and study the encryption algorithm.

Parts I–VII of the assignment involve writing functions to implement the encryption algorithm, while Parts VIII–X involve functions for implementing the decryption algorithm. Some of the earlier functions are also used in implementing decryption.

Let's get to work!

Part I: Map 2D Array Indices to ADFGVX Coordinates

`(char, char) get_adfgvx_coords(int index1, int index2)`

The function takes two integer indices in the range $[0, 5]$ and returns the characters from the string `"ADFGVX"` located at indices `index1` and `index2`.

The function takes the following arguments, in this order:

- `index1`: The first index.
- `index2`: The second index.

Returns in `$v0`:

- The character from `"ADFGVX"` that is at index `index1`.

Returns in `$v1`:

- The character from `"ADFGVX"` that is at index `index2`.

Returns `-1, -1` in `$v0` and `$v1` for error in any of the following cases:

- `index1` < 0 or `index1` > 5
- `index2` < 0 or `index2` > 5

Examples:

Function Call	Return Values
`get_adfgvx_coords(1,4)`	`D, V`
`get_adfgvx_coords(5,3)`	`X, G`
`get_adfgvx_coords(-2,1)`	`-1, -1`
`get_adfgvx_coords(4,7)`	`-1, -1`
`get_adfgvx_coords(8,-3)`	`-1, -1`

Part II: Find a Character's Indices in a 2D Array of Characters

`(int, int) search_adfgvx_grid(char[][] adfgvx_grid, char plaintext_char)`

This function searches the 6×6 array of characters `adfgvx_grid` for `plaintext_char` and returns its 0-based row and columns indices.

The function takes the following arguments, in this order:

- `adfgvx_grid`: A 6×6 2D `char` array containing the 26 letters of the alphabet in uppercase and the digit characters 0-9. The characters are randomly ordered in the grid.
- `plaintext_char`: The character whose 2D indices we want to find.

Returns in `$v0`:

- The row index of `plaintext_char` in `adfgvx_grid`.

Returns in `$v1`:

- The column index of `plaintext_char` in `adfgvx_grid`.

Returns `-1,-1` in `$v0` and `$v1` for error in any of the following cases:

- `plaintext_char` is not found `adfgvx_grid`.

Assumptions:

- You may assume that `adfgvx_grid` is a valid ADFGVX array consisting of 36 unique characters drawn from A-Z, 0-9.

Additional requirements:

- Do not modify the contents of `adfgvx_grid` in memory.

Examples:

In the examples below, `adfgvx_grid` is the 2D grid

```
Q W E 3 R T   A
0 Y U 2 I O   D
4 P L K 9 J   F
1 H G F 5 D   G  <---
S A Z 8 6 X   V      |
C V B 7 N M   X      | coordinates (not part of the grid)
                     |
A D F G V X  <-------
```

which is stored in row-major order as

`"QWE3RT0YU2IO4PLK9J1HGF5DSAZ86XCVB7NM"`

Thus, the letter `Q` is at index `[0][0]` of the array.

Function Call	Return Values
`search_adfgvx_matrix(grid, 'G')`	`3, 2`
`search_adfgvx_matrix(grid, 'X')`	`4, 5`
`search_arfgvx_matrix(grid, '-')`	`-1, -1`
`search_arfgvx_matrix(grid, 'g')`	`-1, -1`

Part III: Map Plaintext Characters to ADFGVX Coordinates

```
void map_plaintext(char[][] adfgvx_grid, String plaintext,
                   char[][] middletext_buffer)
```

This function iterates through each letter in the string `plaintext`, maps each letter to its 2D "ADFGVX coordinates", and then appends the corresponding coordinates to the `middletext_buffer` array.

The function takes the following arguments, in this order:

- `adfgvx_grid`: A 6×6 2D `char` array containing the 26 letters of the alphabet in uppercase and the digit characters 0-9. The characters are randomly ordered in the grid.
- `plaintext`: The null-terminated plaintext message to convert to ADFGVX coordinates. You may assume that the plaintext consists only of uppercase letters and digits.
- `middletext_buffer`: A 2D `char` array that has enough space to hold the mapped plaintext in row-major order characters.

Assumptions:

- You may assume that `adfgvx_grid` is a valid ADFGVX array consisting of 36 unique characters drawn from A-Z, 0-9.

Additional requirements:

- `map_plaintext` must call both `search_adfgvx_grid` and `get_adfgvx_coords`.
- Do not modify the contents of `adfgvx_grid` or `plaintext` in memory.

Example:

To understand the operation of this function, suppose `adfgvx_grid` is the 2D array below, where `I` is at index `[0][0]`:

```
I Q S P 4 T   A
O N L Z J U   D
G A C H X V   F
E 7 3 W K Y   G
5 M B 1 2 6   V
9 8 0 F R D   X

A D F G V X
```

and the plaintext message is `"STONYBROOKUNIV"`. With the help of the `search_adfgvx_grid` and `get_adfgvx_coords` functions, `'S'` is mapped first to `(0,2)` and then to `('A','F')`. We refer to `('A','F')` as the "ADFGVX coordinates" of `'S'`. Characters `'A'` and `'F'` are written to indices `0` and `1` of `middletext_buffer`, respectively. The letter `'T'` is mapped to `(0,5)` and then `('A','X')`. The characters `A` and `X` are written to indices `2` and `3` of `middletext_buffer`, respectively. All other characters of `plaintext` are likewise mapped to their ADFGVX coordinates, which are then appended one pair at a time to `middletext_buffer`.

For this example, suppose `middletext_buffer` were initialized with the contents

```
"????????????????????????????????"
```

After the function has been called, `middletext_buffer` will contain

`"AFAXDADDGXVFXVDADAGVDXDDAAFX???"`

Another example: suppose `adfgvx_grid` is the same as given in the previous example, `plaintext` is `STORMLIGHT` and `middletext_buffer` is initialized with

`"**************************"`

After the function has been called, `middletext_buffer` will contain

`"AFAXDAXVVDDFAAFAFGAX******"`

In both of these examples, `middletext_buffer` was actually larger than it needed to be to store the result. This will not always be the case.

Part IV: Swap the Contents of Two Columns in a 2D Array

```
int swap_matrix_columns(char[][] matrix, int num_rows, int num_cols,
                        int col1, int col2)
```

This function swaps the contents of columns `col1` and `col2` in 2D array `matrix`.

The function takes the following arguments, in this order:

- `matrix`: The array to be modified.
- `num_rows`: Number of rows in `matrix`.
- `num_cols`: Number of columns in `matrix`.
- `col1`: Index of first column to swap.
- `col2`: Index of second column to swap. Available at `0($sp)`.

Note that this function will look at `0($sp)` to find the fifth argument. The *caller* (not `swap_matrix_columns`) is responsible for both allocating and deallocating the four bytes for the fifth argument. See the testing main for this file to see how this is accomplished.

Returns in `$v0`:

- `$v0`: `0` for success, `-1` for an error

Returns `-1` in `$v0` for error in any of the following cases:

- `num_rows` ≤ 0
- `num_cols` ≤ 0
- `col1` < 0 or `col1` $\geq$ `num_cols`
- `col2` < 0 or `col2` $\geq$ `num_cols`

Example:

```
char[][] chars = [
  I Q S P
  4 T O N
  L Z J U
  G A C H
  X V E 7
  3 W K Y
]
swap_matrix_columns(chars, 6, 4, 0, 2)
```

Single quotation marks have been omitted for sake of clarity. After the function call completes, `chars` will be:

```
char[][] chars = [
  S Q I P
  O T 4 N
  J Z L U
  C A G H
  E V X 7
  K W 3 Y
]
```

Part V: Sorting the Columns of a 2D Array

```
void key_sort_matrix(char[][] matrix, int num_rows, int num_cols, T[] key,
                     int elem_size)
```

This function performs an in-place sort (e.g., bubblesort) on `key` and, whenever a swap occurs between elements in `key`, the corresponding columns in `matrix` are also swapped. The notation `T[]` indicates that the `key` could be an array of varying types. For example, if elements `key[2]` and `key[6]` are swapped during a step of sorting, then columns 2 and 6 of `matrix` must also be swapped by calling the `swap_matrix_columns` function. The elements in `key` are compared using their numerical values to determine which is smaller. You may assume that `key`'s length is equal to `num_cols`.

The function takes the following arguments, in this order:

- `matrix`: Matrix that is manipulated as a result of sorting the `key`.
- `num_rows`: The number of rows in the matrix.
- `num_cols`: The number of columns in the matrix.
- `key`: The array of items being sorted.
- `elem_size`: Size in bytes of one element in `key`. Available at `0($sp)`. `elem_size` is guaranteed to be 1 or 4. Elements are unsigned values.

We recommend that you implement a simple algorithm like bubblesort for any sorting required by this assignment:

```
// Assume the array to be sorted is A[] with length n:
for (int i = 0; i < n; i++)
    for (int j = 0; j < n - i - 1; j++)
        if A[j] > A[j + 1]:
            swap A[j] and A[j + 1]
```

Additional requirements:

- This function must call `swap_matrix_columns`.

Example #1. A `key` of characters.

```
matrix = [
   A B C D E F
   G H I J K L
   M N O P Q R
]
key = "WOLFIE"

key_sort_matrix(matrix, 3, 6, key, 1)
```

`matrix` and `key` after the function call completes:

```
matrix = [
   F D E C B A
   L J K I H G
   R P Q O N M
]
key = "EFILOW"
```

Example #2. A `key` of words.

```
matrix = [
   A B C D E
   F G H I J
   K L M N O
   P Q R S T
]
key = [500, 20, 100, 40, 300]

key_sort_matrix(matrix, 4, 5, key, 4)
```

`matrix` and `key` after the function call completes:

```
matrix = [
   B D C E A
   G I H J F
   L N M O K
   Q S R T P
```

```
]
key = [20, 40, 100, 300, 500]
```

Part VI: Matrix Transposition

```
int transpose(char[][] matrix_src, char[][] matrix_dest, int num_rows,
              int num_cols)
```

This function transposes `matrix_src`, storing the result in `matrix_dest`. The values of `matrix_src` are stored in row-major order. You may assume that `matrix_dest` is the same size in bytes as `matrix_src`.

If you are not familiar with matrix transposition, the Wikipedia article on the topic is worth looking at quickly. Briefly, using nested for-loops, each value loaded from `matrix_src[i][j]` is stored in `matrix_dest[j][i]`, where `i` is the row number and `j` is the column number in `matrix_src`. Note that the roles of `i` and `j` are exchanged when a value is written to `matrix_dest`.

The function takes the following arguments, in this order:

- `matrix_src`: A 2D matrix of characters to transpose.
- `matrix_dest`: The buffer in which to store the transposed matrix. The buffer will be large enough to store the result.
- `num_rows`: The number of rows in `matrix_src`.
- `num_cols`: The number of columns in `matrix_src`.

Returns in `$v0`:

- `$v0`: `0` for success, `-1` for an error

Returns `-1` in `$v0` for error in any of the following cases:

- `num_rows` ≤ 0
- `num_cols` ≤ 0

Example #1:

```
matrix_src = [
  A B C D E
  F G H I J
  K L M N O
  P Q R S T
]
transpose(matrix_src, matrix_dest, 4, 5)
```

`matrix_dest` after the function call completes:

```
matrix_dest = [
  A F K P
```

```
  B G L Q
  C H M R
  D I N S
  E J O T
]
```

Example #2:

```
matrix_src = [
  A
  B
  C
  D
  E
  F
]
transpose(matrix_src, matrix_dest, 6, 1)
```

`matrix_dest` after the function call completes:

```
matrix_dest = [
  A B C D E F
]
```

Part VII: Encryption of a Plaintext Message

```
void encrypt(char[][] adfgvx_grid, String plaintext, String keyword,
             char[] ciphertext)
```

This function takes the null-terminated string `plaintext` and stores it sequentially in the `ciphertext` string, null-terminating the ciphertext.

The function takes the following arguments, in this order:

- `adfgvx_grid`: A 6×6 2D `char` array containing the 26 letters of the alphabet in uppercase and the digit characters 0-9. The characters are randomly ordered in the grid.
- `plaintext`: The plaintext message to be encrypted. You may assume that the plaintext will consist only of uppercase letters and digit characters.
- `keyword`: The keyword being used to encrypt the message. You may assume that the keyword will consist only of uppercase letters and digit characters.
- `ciphertext`: A buffer that will contain the encrypted ciphertext. The buffer is guaranteed to be large enough to store the null-terminated encrypted message.

Assumptions:

- You may assume that `adfgvx_grid` is a valid ADFGVX array consisting of 36 unique characters drawn from A-Z, 0-9.

Additional requirements:

- `encrypt` must call `map_plaintext`, `key_sort_matrix`, and `transpose` as illustrated in the examples below.
- Do not modify the contents of `adfgvx_grid` in memory.

Example #1: Simple (Special) Case

Variable initialization:

```
adfgvx_grid = [
   N A 1 C E H
   8 T B 2 O M
   3 5 W R P D
   4 F 6 G 7 I
   9 J 0 K L Q
   S U V X Y Z
]
keyword = "KNIGHTS"
plaintext = "THEWAYOFKINGSLIFEBEFOREDEATH"
```

The first function call we make is to `map_plaintext` to map each plaintext character to a pair of ADFGVX characters. However, where are we going to store the result? We will allocate some memory on the *heap* using system call #9, which is called sbrk. For example, to allocate 56 bytes of memory we would write this code:

```
li $a0, 56
li $v0, 9
syscall
```

When the system call completes, the address of the newly-allocated memory buffer will be available in `$v0`. The address will be on a word-aligned boundary, regardless of the value passed through `$a0`. Unfortunately, there is no way in MARS to de-allocate memory to avoid creating memory leaks. The run-time systems of Java, Python and some other languages take care of freeing unneeded memory blocks with garbage collection, but assembly languages, C/C++, and other languages put the burden on the programmer to manage memory. You will learn more about this in CSE 320.

So let's suppose now that we have performed the system call to allocate enough memory to store the array off ADFGVX characters. (You will need to figure out how many bytes to store.) Let's call this array `heap_ciphertext_array`.

For reasons that will be clearer in Example #2, we will fill the array with asterisks (write a loop or a helper function for yourself):

```
"**********************************************************"
```

Now we can use the starting address of `heap_ciphertext_array` as the third argument to `map_plaintext`. After the call to `map_plaintext` returns, `heap_ciphertext_array` should have the following contents:

```
"DDAXAVFFADXVDVGDVGGXAAGGXAVVGXGDAVDFAVGDDVFGAVFXAVADDDAX"
```

If we treat this 1D array as a 2D array of characters stored in row-major-order with 8 rows and 7 columns (ask yourself: why that number of rows and columns?), we have this:

```
D D A X A V F
F A D X V D V
G D V G G X A
A G G X A V V
G X G D A V D
F A V G D D V
F G A V F X A
V A D D D A X

K N I G H T S  <-- key (not part of heap_ciphertext_array)
```

Next we make a function call to `key_sort_matrix` to sort `heap_ciphertext_array` using `keyword` as the sorting key. `heap_ciphertext_array` will be transformed into:

```
X A A D D F V
X V D F A V D
G G V G D A X
X A G A G V V
D A G G X D V
G D V F A V D
V F A F G A X
D D D V A X A

G H I K N S T  <-- sorted keyword
```

The letters of the ciphertext need to be read out in column-by-column fashion by the algorithm, but to make things easier for ourselves for debugging and processing, we will transpose the 2D array, thereby converting it into row-by-row order. This will let us read out the characters from index 0 to the final index in sequential order. Note that if the length of the plaintext were not a multiple of the length of the keyword, there would be at least one column shorter than the others. We will see this in the next example.

To transform the matrix as described in the previous paragraph, call the `transpose` function, using `ciphertext` as the destination matrix. `ciphertext` will then contain:

```
X X G X D G V D
A V G A A D F D
A D V G G V A D
D F G A G F F V
D A D G X A G A
F V A V D V A X
V D X V V D X A
```

If we read out the characters of `ciphertext` in row-by-row (sequential) order, we have the string:

```
"XXGXDGVDAVGAADFDADVGGVADDFGAGFFVDADGXAGAFVAVDVAXVDXVVDXA"
```

which is the encrypted message. As a final step we need to null-terminate the `ciphertext` string.

Example #2: General Case

```
adfgvx_grid = [
   N A 1 C E H
   8 T B 2 O M
   3 5 W R P D
   4 F 6 G 7 I
   9 J 0 K L Q
   S U V X Y Z
]
keyword = "KNIGHTS"
plaintext = "THEIMMORTALWORDS"
```

First we need to allocate memory for `heap_ciphertext_array`. For this particular example, where the keyword is 7 characters in length and the plaintext is 16 characters in length, we need a `heap_ciphertext_array` of 35 bytes. To see why, follow this example to the end.

After allocating the memory for `heap_ciphertext_array` and filling it with asterisks, we make the call to `map_plaintext` as described in the previous example. `heap_ciphertext_array` will have the following contents:

```
"DDAXAVGXDXDXDVFGDDADVVFFDVFGFXXA***"
```

If we interpret `heap_ciphertext_array` as a 2D array of characters with 5 rows and 7 columns stored in row-major order, we have this:

```
D D A X A V G
X D X D X D V
F G D D A D V
V F F D V F G
F X X A * * *

K N I G H T S  <-- keyword (not part of heap_ciphertext_array)
```

Before continuing, stop reading and figure out for yourself why the array has 5 rows and 7 columns.

Next we make the function call to `key_sort_matrix`. `heap_ciphertext_array` will be transformed into:

```
X A A D D G V
D X X X D V D
D A D F G V D
D V F V F G F
A * X F X * *

G H I K N S T  <-- sorted keyword
```

Next we call `transpose` on `heap_ciphertext_array` and write the result into `ciphertext`, which be-

comes:

```
X D D D A
A X A V *
A X D F X
D X F V F
D D G F X
G V V G *
V D D F *
```

Now if we read out the characters of `ciphertext` in row-by-row (sequential) order, we have the encrypted message:

`"XDDDAAXAV*AXDFXDXFVFDDGFXGVVG*VDDF*"`

Example #3: A Keyword Containing Digits

If `adfgvx_grid` is as above, the keyword is `"SBU2018NY"` and the plaintext is `"CSE220LOVESMIPS"`, the ciphertext is `"AVD*VDX*AVA*DVG*GVVXGXX*AGAVXFXADFF*"`.

Part VIII: Decrypting Character Pairs

`(int, char) lookup_char(char[][] adfgvx_grid, char row_char, char col_char)`

The first step to decrypting ciphertext begins with being able to take any pair of ADFGVX characters and convert them back to the correct plaintext character using the jumbled 6×6 ADFGVX character matrix.

This function takes the following arguments, in this order:

- `adfgvx_grid`: A 6×6 2D `char` array containing the 26 letters of the alphabet in uppercase and the digit characters 0-9. The characters are randomly ordered in the grid.
- `row_char`: The ADFGVX character to serve as the row "coordinate".
- `col_char`: The ADFGVX character to serve as the column "coordinate".

Returns in `$v0`:

- `0` if the look-up is successful, `-1` for an error

Returns in `$v1`:

- The char from `adfgvx_grid` found at ADFGVX coordinates `(row_char, col_char)`, if the lookup was successful. Any value can be returned in `$v1` on error.

Returns `-1` in `$v0` for error in any of the following cases:

- `row_char` is not one of A, D, F, G, V or X
- `col_char` is not one of A, D, F, G, V or X

Assumptions:

- You may assume that `adfgvx_grid` is a valid ADFGVX array consisting of 36 unique characters drawn from A-Z, 0-9.

Example

Suppose we have the following arguments:

```
adfgvx_grid = [
  Q W E 3 R T
  0 Y U 2 I O
  4 P L K 9 J
  1 H G F 5 D
  S A Z 8 6 X
  C V B 7 N M
]
row_char = 'F'
col_char = 'G'
```

The return values in this case will be `(0, 'K')`.

Part IX: Sorting a String

```
void string_sort(String str)
```

This function performs an in-place sort of the null-terminated string `str`, using character ASCII codes as the keys. We recommend you implement bubblesort or a similar, simple algorithm.

The function takes one argument:

- `str`: The string to sort.

Function Call	Resulting `str` Value
`string_sort("KNIGHTS")`	`"GHIKNST"`
`string_sort("D")`	`"D"`

Part X: Decryption of Ciphertext

```
void decrypt(char[][] adfgvx_grid, String ciphertext, String keyword,
             String plaintext)
```

This function takes a fully encrypted message, decrypts it, and saves the decrypted message in `plaintext`. To perform decryption we must call the functions `string_sort`, `transpose`, `key_sort_matrix`, and `lookup_char` in a particular order, which is illustrated by way of example below.

The function takes the following arguments, in this order:

- `adfgvx_grid`: A 6×6 2D `char` array containing the 26 letters of the alphabet in uppercase and the digit characters 0-9. The characters are randomly ordered in the grid.
- `ciphertext`: The null-terminated ciphertext to be decrypted. You may assume that the ciphertext is valid.
- `keyword`: The keyword used to encrypt the original plaintext. You may assume that this is the correct keyword that was used to encrypt the original plaintext.
- `plaintext`: Space allocated to save the null-terminated decrypted ciphertext. You may assume that the buffer is large enough to store the null-terminated result.

Additional requirements:

- `decrypt` must call `string_sort`, `transpose`, `key_sort_matrix`, and `lookup_char`.

Example:

Variable initialization:

```
adfgvx_grid = [
   N A 1 C E H
   8 T B 2 O M
   3 5 W R P D
   4 F 6 G 7 I
   9 J 0 K L Q
   S U V X Y Z
]
keyword = "KNIGHTS"
ciphertext = "XDDDAAXAV*AXDFXDXFVFDDGFXGVVG*VDDF*"
```

We begin by using `sbrk` to allocate some memory on the heap to store a copy of the keyword. Let's refer to this array as `heap_keyword`. Then we sort `heap_keyword` by calling `string_sort` on it. In this example, `heap_keyword` will become `"GHIKNST"`.

Then we use `sbrk` again to allocate memory for an array called `heap_keyword_indices` that contains a number of unsigned integers equal to the length of `keyword`. In this example, `heap_keyword_indices` will be an array of 7 integers:

```
heap_keyword_indices = new int[7]
```

The purpose of this array will be to help "undo" the sorting step of the encryption process. To undo the sorting, first we need to figure out where each letter of the original keyword wound up in the sorted keyword. Here's the pseudocode:

```
for (int i = 0; i < i len(keyword); i++)
    heap_keyword_indices[i] = keyword.index_of(heap_keyword[i])
```

`index_of` is intended as a helper function that returns the 0-based index of a character in a string. After this loop has completed, `heap_keyword_indices` for this example will contain:

```
heap_keyword_indices = [3, 4, 2, 0, 1, 6, 5]
```

As we can see, this array tells us where each character in the sorted keyword was originally found in the unsorted keyword. For example, `G` at index 0 in the sorted keyword was originally at index 3 in the unsorted keyword. `H` at index 1 in the sorted keyword was originally at index 4 in the unsorted keyword. The `heap_keyword_indices` will tell us how to "unsort" the ciphertext when treated as a 2D array. Recall:

```
ciphertext = "XDDDAAXAV*AXDFXDXFVFDDGFXGVVG*VDDF*"
```

If we interpret the contents of `ciphertext` as a 2D array with 7 rows and 5 columns (why? how do we figure out those values?) we have:

```
X D D D A
A X A V *
A X D F X
D X F V F
D D G F X
G V V G *
V D D F *
```

Next we need to transpose this array. Use `sbrk` to allocate a 2D array on the heap to store the transpose. Let's call this array `heap_ciphertext_array`, which after the call to `transpose` will contain:

```
X A A D D G V
D X X X D V D
D A D F G V D
D V F V F G F
A * X F X * *
```

Now we can finally "unsort" the matrix by calling the `key_sort_matrix` function with `heap_ciphertext_array` for the `matrix` argument and `heap_keyword_indices` for the `key` argument. `heap_keyword_indices` and `heap_ciphertext_array` will change to the following:

```
heap_keyword_indices =  [0, 1, 2, 3, 4, 5, 6]
heap_ciphertext_array = [
   D D A X A V G
   X D X D X D V
   F G D D A D V
   V F F D V F G
   F X X A * * *
]
```

We're almost there! The ciphertext is now in a form that will be easy to decode from `heap_ciphertext_array`: it's in row-major-order and we have recovered the original order of the rows of ADFGVX coordinates. We now need to write a loop that will iterate over `heap_ciphertext_array`, treating it like a 1D array of characters. The loop should read out the ADFGVX characters as pairs (`DD`, `AX`, `AV`, etc.), decode them using `lookup_char`, and append the decrypted characters one-by-one to `plaintext`. After appending the null-terminator, we produce the original plaintext:

```
"THEIMMORTALWORDS"
```

Congratulations! You have implemented ADFGVX encryption and decryption in MIPS assembly language!

Academic Honesty Policy

Academic honesty is taken very seriously in this course. By submitting your work to Blackboard you indicate your understanding of, and agreement with, the following Academic Honesty Statement:

1. I understand that representing another person's work as my own is academically dishonest.
2. I understand that copying, even with modifications, a solution from another source (such as the web or another person) as a part of my answer constitutes plagiarism.
3. I understand that sharing parts of my homework solutions (text write-up, schematics, code, electronic or hard-copy) is academic dishonesty and helps others plagiarize my work.
4. I understand that protecting my work from possible plagiarism is my responsibility. I understand the importance of saving my work such that it is visible only to me.
5. I understand that passing information that is relevant to a homework/exam to others in the course (either lecture or even in the future!) for their private use constitutes academic dishonesty. I will only discuss material that I am willing to openly post on the discussion board.
6. I understand that academic dishonesty is treated very seriously in this course. I understand that the instructor will report any incident of academic dishonesty to the College of Engineering and Applied Sciences.
7. I understand that the penalty for academic dishonesty might not be immediately administered. For instance, cheating in a homework may be discovered and penalized after the grades for that homework have been recorded.
8. I understand that buying or paying another entity for any code, partial or in its entirety, and submitting it as my own work is considered academic dishonesty.
9. I understand that there are no extenuating circumstances for academic dishonesty.

How to Submit Your Work for Grading

To submit your `hw3.asm` file for grading:

1. Login to Blackboard and locate the course account for CSE 220.
2. Click on "Assignments" in the left-hand menu and click the link for this assignment.
3. Click the "Browse My Computer" button and locate the `hw3.asm` file. Submit only that one `.asm` file.
4. Click the "Submit" button to submit your work for grading.

Oops, I messed up and I need to resubmit a file!

No worries! Just follow steps 4–6 again. We will grade only your last submission.