CSE 220: Systems Fundamentals I

Stony Brook University

Programming Project #2

Fall 2019

Assignment Due: Friday, October 25, 2019 by 11:59 pm

Updates to the Document:

- 10/14/2019: Part IV: `alphabet` is null-terminated
- 10/14/2019: Part V: `alphabet` is null-terminated
- 10/14/2019: Part VI: `base64_table` is null-terminated
- 10/14/2019: Part VII: `base64_table` is null-terminated
- 10/14/2019: Part VIII: `key_square` is null-terminated
- 10/14/2019: Part IX: `key_square` is null-terminated

Learning Outcomes

After completion of this programming project you should be able to:

- Read and write strings of arbitrary length.
- Implement non-trivial algorithms that require conditional execution and iteration.
- Design and code functions that implement the MIPS assembly function calling conventions.

Getting Started

Visit Piazza and download the file `proj2.zip`. Decompress the file and then open `proj2.zip`. Fill in the following information at the top of `proj2.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 `proj2.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 `proj2.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 into 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 `proj2.asm` file. A submission that contains a `.data` section will probably receive a score of zero.

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.
- **You must use the Stony Brook version of MARS posted on Blackboard.** 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 might need to complete the homework assignments.
- 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 the programming projects 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.
- Do not submit a file with the function/label `main` 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 assignment 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:

- Each test case 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. To find the instruction count of your code in MARS, go to the **Tools** menu and select **Instruction Statistics**. Press the button marked **Connect to MIPS**. Then assemble and run your code as normal.

- Any excess output from your program (debugging notes, etc.) might impact grading. Do not leave erroneous print-outs in your code.
- To avoid side-effects caused by system calls, the grading scripts will comment-out any instances of `syscall` found in your code. Make sure you test your code with this in mind.
- 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.

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 really 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 `$v` registers as a means of "helping" the caller.
- "Hiding" values in the `$k`, `$f` and `$at` registers or storing values in main memory by way of offsets to `$gp`. This is basically cheating or at best, a form of laziness, so don't do it. We will comment out any such code we find.

How to Test Your Functions

To test your implementated functions, open the provided `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 `proj2.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, create your own test cases in those main files. Your submission will not be graded using the examples provided in this document or using the provided main file(s). Do not submit your main files to Blackboard – we will delete them.

Again, any modifications to the `main` files will not be graded. You will submit only your `proj2.asm` for grading. Make sure that all code required for implementing your functions is included in the `proj2.asm` file. To make sure that your code is self-contained, try assembling your `proj2.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 `proj2.asm`.

A Reminder on How Your Work Will be Graded

It is **imperative** (crucial, essential, necessary, critically important) that you implement the functions below exactly as specified. Do not deviate from the specifications, even if you think you are implementing the program in a better way. Modify the contents of memory only as described in the function specifications!

Basic Encryption Algorithms

In this assignment you will be implementing three different encryption/encoding schemes: a simple substitution cipher, the bifid cipher, and Base64 encoding. To support the implementation of these algorithms we first need to write a few utility functions.

Part I: Compute a String's Length

```
int strlen(string str)
```

This function takes a null-terminated string (possibly empty) and returns its length (i.e., the number of characters in the string).

The function takes the following arguments, in this order:

- `str`: The starting address of a null-terminated string

Returns in `$v0`:

- The length of the string, not including the null-terminator.

Additional requirements:

- The function must not write any changes to main memory.

Examples:

Function Arguments	Return Value
"**[CSE 220 Fall 2019]!!!**\0"	26
"stonybrook\0"	10
"\0"	0

Part II: Find the Index of a Character in a String

`int index_of(string str, char ch)`

The function returns the index of the first instance of character `ch` in the null-terminated string `str`, starting at index 0. If the character is not present in the string, the function returns `-1`. It is possible that the string is empty.

The function takes the following arguments, in this order:

- `str`: The starting address of a null-terminated string.
- `ch`: The character to search for.

Returns in `$v0`:

- The index of the leftmost occurrence of `ch` in `str`, if any, or `-1` if `ch` is not present in `str`.

Additional requirements:

- The function must not write any changes to main memory.

Examples:

Function Arguments	Return Value
"**[CSE 220 Fall 2019]!!!**\0", 'F'	11
"**[CSE 220 Fall 2019]!!!**\0", '0'	9
"\0", 'A'	-1

Part III: Copy an Array of Bytes

`int bytecopy(char[] src, int src_pos, char[] dest, int dest_pos, int length)`

Modeled after Java's `System.arraycopy` method, the `bytecopy` function copies `length` bytes from array `src`, starting at index `src_pos` and writes them in array `dest`, starting at index `dest_pos`. `src` and `dest` are not necessarily null-terminated.

If `length` is less than or equal to zero, the function writes no changes to memory and returns `-1`. If either or both of `src_pos` and `dest_pos` is less than zero, the function writes no changes to memory and returns `-1`.

The function does not need to consider the possibility that it might overrun the end of the `dest` array or that it will run out of characters while reading the `src` array. Doing such checks would be the responsibility of the caller.

The function takes the following arguments, in this order:

- `src`: The starting address of an array of bytes that the function reads from.
- `src_pos`: The index of `src` to start reading characters from.
- `dest`: The starting address of an array of bytes that the function writes to.
- `dest_pos`: The index of `dest` to start writing characters to.
- `length`: The number of characters to copy from `src` to `dest`.

Returns in `$v0`:

- The number of characters copied from `src` to `dest`, or `-1` on error.

Additional requirements:

- The function must not write any changes to main memory except as specified. For example, the function must not null-terminate the `dest` array.

Examples:

In the examples below we will use the following byte arrays (strings):

```
src = "ABCDEFGHIJKLMNOP"
dest = "abcdefghijklmn"
```

Example #1:

```
src_pos = 3
dest_pos = 4
length = 5
```

Contents of `dest` after the function call: `"abcdDEFGHjklmn"`

Return value: `5`

Example #2:

```
src_pos = 3
dest_pos = 0
length = 5
```

Contents of `dest` after the function call: `"DEFGHfghijklmn"`

Return value: `5`

Example #3:

```
src_pos = -2
dest_pos = 4
length = 5
```

Contents of `dest` after the function call: `"abcdefghijklmn"`

Return value: `-1`

Part IV: Encrypt a Message using a Simple Substitution Cipher

```
int scramble_encrypt(char[] ciphertext, string plaintext, string alphabet)
```

This function implements a simple substitution cipher that encrypts each letter of some plaintext with a different character according to a 52-character *substitution alphabet* provided as the third argument to the function. This character array serves as the key of the encryption algorithm. The resulting encrypted text (the ciphertext) is written to the memory region identified by the first argument. In this cipher, the first 26 letters of `alphabet` indicate the substitutions for the uppercase letters, and the second group of 26 letters of `alphabet` provide the substitutions for the lowercase letters. The null-terminated `alphabet` argument will always be 52 characters in length (not including its null-terminator) and contain exactly one instance of each letter in both uppercase and lowercase. This cipher does not encrypt non-letter characters (i.e., digits, spaces, punctuation marks, etc.), but merely copies them unchanged to the ciphertext. After writing the ciphertext, the function must null-terminate the ciphertext. The function returns the number of letter characters it encrypted. The `plaintext` argument is guaranteed to be null-terminated.

As an example, suppose the substitution alphabet, given by the argument `alphabet`, is:

```
"QeKEPOslaJbkfxUDdGTIStNwhjXnYCLvRpyFqBzmAuHrgoiZMcWV"
```

This substitution alphabet indicates that a capital letter **A** from the plaintext will be replaced with an uppercase **Q** in the ciphertext, a capital letter **B** will be replaced with a lowercase **e**, and so on. Continuing to the 27th letter of `alphabet`, a lowercase **a** in the plaintext will be replaced with a capital **X**, a lowercase **b** with a lowercase **n**, and so on:

```
plaintext letter: ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
replacement:      QeKEPOslaJbkfxUDdGTIStNwhjXnYCLvRpyFqBzmAuHrgoiZMcWV
```

The function takes the following arguments, in this order:

- `ciphertext`: The starting address of a memory buffer where the encrypted message is written. The buffer will always be at least large enough to store the ciphertext, including a null-terminator.
- `plaintext`: The starting address of a null-terminated string to encrypt. It is possible that the plaintext is empty.
- `alphabet`: The 52-character, null-terminated string of uppercase and lowercase letters used to encrypt the ciphertext.

Returns in `$v0`:

- The number of letter characters that were encrypted.

Additional requirements:

- The function must not write any changes to main memory except as specified.

Example:

```
plaintext = "Do your homework and study! CSE 220, SBU, Fall 2019!!!\0"
alphabet =  "QeKEPOslaJbkfxUDdGTIStNwhjXnYCLvRpyFqBzmAuHrgoiZMcWV"
```

Expected `ciphertext`:

```
"EA WAir pAzLMArq XmC goiCW! KTP 220, TeS, OXBB 2019!!!\0"
```

Return value: `32`

Part V: Decrypt a Message Encrypted by a Simple Substitution Cipher

```
int scramble_decrypt(char[] plaintext, string ciphertext, string alphabet)
```

This function decrypts the `ciphertext` string that was encrypted using the substitution cipher described in the previous part of the assignment. As in the `scramble_encrypt` function, the first 26 letters of `alphabet` indicate the substitutions for the uppercase letters, and the second group of 26 letters of `alphabet` provide the substitutions for the lowercase letters. The null-terminated `alphabet` argument will always be 52 characters in length (not including its null-terminator) and contain exactly one instance of each letter in both uppercase and lowercase. The function uses the alphabet to decrypt the ciphertext by performing a look-up into the alphabet to determine the original plaintext characters, writing them into `plaintext`. The function copies non-letters unchanged from `ciphertext` to `plaintext`. After writing the plaintext, the function null-terminates `plaintext`. The function returns the number of letter characters it decrypted. The `ciphertext` argument is guaranteed to be null-terminated.

The function takes the following arguments, in this order:

- `plaintext`: The starting address of a memory buffer where the decrypted message is written. The buffer will always be at least large enough to store the plaintext, including a null-terminator.
- `ciphertext`: The starting address of a null-terminated string to decrypt. It is possible that the ciphertext is empty.
- `alphabet`: The 52-character, null-terminated string of uppercase and lowercase letters used to decrypt the ciphertext.

Returns in `$v0`:

- The number of letter characters that were decrypted.

Additional requirements:

- The function must not write any changes to main memory except as specified.
- `scramble_decrypt` must call `index_of`.

Example:

```
ciphertext = "EA WAir pAzLMArq XmC goiCW! KTP 220, TeS, OXBB 2019!!!\0"
alphabet =   "QeKEPOslaJbkfxUDdGTIStNwhjXnYCLvRpyFqBzmAuHrgoiZMcWV"
```

Expected `plaintext`:

`"Do your homework and study! CSE 220, SBU, Fall 2019!!!\0"`

Return value: `32`

Part VI: Base64 Encoding of Strings

`int base64_encode(char[] encoded_str, string str, string base64_table)`

Before reading further, read the first half of Wikipedia article on Base64 encoding. You can stop reading when you get to the section labeled "Implementations and history".

This function takes a non-empty, null-terminated string `str` and encodes it using Base64 encoding, writing the result as the null-terminated string `encoded_str`. The the null-terminator at the end of `str` is *not* encoded in `encoded_str`. As explained in the linked Wikipedia article, each 8-bit ASCII character of `str` is encoded as a 6-bit code taken from the 64-character array `base_table`. Although the characters in `base_table` are the same as those used in the normal Base64 encoding algorithm given in the Wikipedia article, the characters themselves may appear in any order inside `base_table`. Examples below will help to clarify what we mean here. Also as explained in the linked article, one or two equal signs sometimes must appended to the end of the encoded string so that the number of characters in the encoded string is a multiple of 4. The function null-terminates `encoded_str` before returning.

- `encoded_str`: The starting address of a memory buffer where a Base64-encoded string computed by this function will be stored. The buffer will be large enough to accommodate a null-terminator as well at the end of the encoded message.
- `str`: A null-terminated string to be encoded.
- `base64_table`: The 64-character null-terminated string of uppercase and lowercase letters, digits and miscellaneous symbols used to encode the string `str`.

Returns in `$v0`:

- The length of the encoded string, not including the null-terminator.

Additional requirements:

- The function must not write any changes to main memory except as specified.

Example #1:

```
str = "Let's hear it for MIPS! Goooooo MIPS!\0"
base_table = "HWegnLO7mM0Q6XwTplBduqSNJsZhkrf25cUoE49bAvPtz8IKDyVjiYC31/+axFRG"
```

Expected `encoded_str: "dOqiM365ZOqckUWvreW9h3m5du4puVn5lCFKhCFKhVWXBqWdmp==\0"`

The value of `base_table` given above corresponds with the following table for use in Base64 encoding:

Index	Binary	Char	Index	Binary	Char	Index	Binary	Char	Index	Binary	Char
0	000000	H	16	010000	p	32	100000	5	48	110000	D
1	000001	W	17	010001	l	33	100001	c	49	110001	y
2	000010	e	18	010010	B	34	100010	U	50	110010	V
3	000011	g	19	010011	d	35	100011	o	51	110011	j
4	000100	n	20	010100	u	36	100100	E	52	110100	i
5	000101	L	21	010101	q	37	100101	4	53	110101	Y
6	000110	O	22	010110	S	38	100110	9	54	110110	C
7	000111	7	23	010111	N	39	100111	b	55	110111	3
8	001000	m	24	011000	J	40	101000	A	56	111000	1
9	001001	M	25	011001	s	41	101001	v	57	111001	/
10	001010	0	26	011010	Z	42	101010	P	58	111010	+
11	001011	Q	27	011011	h	43	101011	t	59	111011	a
12	001100	6	28	011100	k	44	101100	z	60	111100	x
13	001101	X	29	011101	r	45	101101	8	61	111101	F
14	001110	w	30	011110	f	46	101110	I	62	111110	R
15	001111	T	31	011111	2	47	101111	K	63	111111	G

The figure below illustrates how the first three characters of `str` are encoded as `dOqi` using the above table:

Source	Text (ASCII)	L							e								t							
	Octets	76 (0x4c)							101 (0x65)								116 (0x74)							
Bits		0	1	0	0	1	1	0	0	0	1	1	0	0	1	0	1	0	1	1	1	0	1	0	0
Base64 encoded	Sextets	19							6					21				52							
	Character	d							O					q				i							
	Octets	100 (0x64)							79 (0x4f)					113 (0x71)				105 (0x69)							

Note how a sextet is used as an index into `base_table` to find the corresponding ASCII symbol. The 8-bit ASCII code for that symbol is then written into `encoded_str`.

Example #2:

```
str = "23 + 52 equals 75\0"
base_table = "3YbGHP1qIX5OyFsxgWKliB7h8AanUk6trD4w+LR0jMZfTp29J/NuCdomvzVcEeSQ"
```

Expected `encoded_str: "ywyr5N3dy4YLUhBDnqyrFui=\0"`

Example #3:

```
str = "Stony Brook University 2019\0"
base_table = "jm3MX7T1wFRHLfVDQlGvtNs/ZCon8JyBig+2EurkPU0K9q4aOhSIW5pxY6Adzceb"
```

Expected `encoded_str: "txlankEiQkFanp9iNs6UJrNS8puWyGjSLMX6\0"`

Part VII: Decoding of Base64-encoded Strings

```
int base64_decode(char[] decoded_str, string encoded_str, string base64_table)
```

This function takes a non-empty, null-terminated string `encoded_str`, which is a string of Base64-encoded data, and decodes it, writing the result as the null-terminated string `encoded_str`. The assumption always is that the original data was a string. The function must null-terminate `decoded_str` before returning.

The function may assume that the original data was encoded with any necessary padding (= symbols) appended to the encoded string. Therefore, the length of `encoded_str` is always guaranteed to be a multiple of 4.

- `decoded_str`: The starting address of a memory buffer where a the function writes the decoded string.
- `encoded_str`: A null-terminated string to be decoded.
- `base64_table`: The 64-character null-terminated string of uppercase and lowercase letters, digits and miscellaneous symbols used to decode the string `encoded_str`.

Returns in `$v0`:

- The length of the decoded string, not including the null-terminator.

Additional requirements:

- The function must not write any changes to main memory except as specified.
- `base64_decode` must call `index_of`.

Example #1:

```
encoded_str = "bYKLodGUOjH8Hw==\0"
base_table = "w964Nem/zAR37BtJxKv0byjkgMOfoH2PqIulnDdT8CV1UcEL5QXhrGsYaZpFi+SW"
```

Expected `decoded_str: "Stormlight\0"`

Example #2:

```
encoded_str = "MipF4izIuMxtcrcy4iwvcNkOANIg+S5/CUn=\0"
base_table = "VxZDRbN7s6T9om8lP5MXwkrS+cJC4AuKv2tYeIiW0/EG3LgyQfBOhFUjnzHdqpa1"
```

Expected `decoded_str: "Journey before destination\0"`

Example #3:

```
encoded_str = "kkV=\0"
base_table = "JacdT30UI79KV1povPfwrkjlZznDYELQGH/Ohm6BXu2itbe8xRWsgC5q4SyAM+FN"
```

Expected `decoded_str: "US\0"`

Part VIII: Encrypt a Message using a Bifid Cipher

```
int bifid_encrypt(char[] ciphertext, string plaintext, string key_square,
                  int period, byte[][] index_buffer, char[][] block_buffer)
```

For this part and the next part of the assignment you will implement a variant of the bifid cipher, but using a 9×9

key square instead of the traditional 5×5 key square. Before reading further, read the linked Wikipedia article on the bifid cipher.

This function implements a bifid-style cipher with a 9×9 key square like the following:

	0	1	2	3	4	5	6	7	8
0	g	.	W	D	O	?	s	1	H
1	t	j	2	X	#	y	d	h	P
2	$	!	o	6	M	-	space	Z	p
3	l	z	\'	a	;	n	V	T	v
4	3	Q	e	k	7	S	B	I	J
5	=	8	C	,	c	u	@	9	/
6	Y	m	4	F	x	b	5	A	K
7	f	(	*	R	U	\"	r	q	)
8	%	w	E	N	:	0	L	i	G

The numbers along the top row and leftmost column provide the "indices" of each character in the table. For example, the uppercase R character's indices are (7, 3). During encryption, each character of the input is replaced by its indices, with the row index written over the column index, as shown in Step 1, below. Then in Step 2 the indices are grouped into blocks of a certain width, known as the *period*. In Step 3 these groups are read off left-to-right, starting at the first row of the block and continuing to the second row of the block. Note that if the length of the plaintext is not a multiple of the period there will be "leftover" indices that form a smaller block than normal. In Step 4, adjacent values in the blocks are taken as pairs of indices. The character at each successive index-pair is appended to the ciphertext. The function null-terminates the ciphertext before returning. The function returns the number of blocks (complete or partial blocks) created during the execution of the algorithm.

```
period: 7
plaintext:  "The Man", The Killers (2017)    [outer quotation marks omitted]

Step 1: row 73142233752314268334702718047
        col 57726435536772687002666125748

Step 2:     7314223 3752314 2683347 0271804 7
            5772643 5536772 6870026 6612574 8

Step 3:     73142235772643 37523145536772 26833476870026 02718046612574 78

Step 4:     R # o n q _ k  T C z S , A *  _ N ; r i g _  W ( % B m - U  )
```

In Step 4, underscores have been used to visualize spaces. The true ciphertext contains spaces, as can be seen below.

Final ciphertext: `R#onq kTCzS,A* N;rig W(%Bm-U)\0` (outer quotation marks omitted for clarity)

Return value: `5` (Why? Because in Step 2 we created four complete blocks and one partial block.)

The function takes the following arguments, in this order:

- `ciphertext`: The starting address of a memory buffer where the null-terminated ciphertext is written.

- `plaintext`: The starting address of a null-terminated string to encrypt. The plaintext is guaranteed to contain only those characters that appear in the key square.
- `key_square`: The 81-character, 2D array of uppercase letters, lowercase letters, digits and other ASCII characters used to encrypt the ciphertext. Characters are stored in row-major-order. For your convenience, `key_square` is null-terminated.
- `period`: The period of the cipher, i.e., how many pairs of indices are taken at a time and mapped to 2D coordinates. If `period` is negative or zero, the function returns `-1` and makes no changes to memory.
- `index_buffer`: A region of memory exactly `2 * strlen(plaintext))` bytes in size that can be used to temporarily store 8-bit unsigned integers. The address of this buffer is passed to the function at memory address `0($sp)`.
- `block_buffer`: A region of memory exactly `2 * period` bytes in size that can be used to temporarily store ASCII characters. The address of this buffer is passed to the function at memory address `4($sp)`.

The function allocates no new memory whatsoever. Any `.data` sections and `syscall` instructions will be deleted from your code before your work is graded. The buffers `index_buffer` and `block_buffer` are provided for sake of convenience to the programmer. You are not required to use either buffer if you don't want to.

Returns in `$v0`:

- The number of blocks of indices created by the algorithm, or `-1` on error when the period is negative or zero.

Additional requirements:

- The function must not write any changes to main memory except as specified.
- `bifid_encrypt` must call `index_of` and `bytecopy`.

Examples

In the examples below, the outermost set of double quotation marks that are typically used indicate a string have been deliberately omitted for sake of clarity.

Example #1:

```
plaintext =  Stony Brook University\0
key_square = #8yszp,gv4\"huWA!w=lMRatXrEDGeBm5j$Z1Iq?Q*n;    (cont. next line)
             PTNV2iC9.H)k06o(f3@d7\'S:%OYK-xL/cUJ bF
period = 6
```

Expected `ciphertext = S(vtjrB3drm,UNBn1!plm?\0`

Expected return value: `4`

Example #2:

```
plaintext =  Computer Science\0
```

```
key_square = 68*be/x!1uSvE0t?U;$l-ci(4nak=)2CW5\'ZOHPDAw   (cont. next line)
             f7qIXs#mjF\"VrLN@3p:BRJzdygGhTYMoQ9 %.,K
period = 2
```

Expected `ciphertext = ZHFwS/xOoH-C*7$C\0`

Expected return value: `8`

Example #3:

```
plaintext =  Dinner @ $50 per person\0
key_square = DQn'v)t;iGI%5dOr*Rbu4zH@xcZ?:/M1XK,0P$9f=m   (cont. next line)
             eps(#VUg aBYCSj"!-N.TlAkWLyh7FE3q2wo8J6
period = 200
```

Expected `ciphertext = DD$Vg5g$g$sDqx- 5oh-hTV\0`

Expected return value: `1`

Part IX: Decrypt a Message Encrypted by a Bifid Cipher

```
int bifid_decrypt(char[] plaintext, string ciphertext, string key_square,
                  int period, byte[][] index_buffer, char[][] block_buffer)
```

This function decrypts the string `ciphertext`, which was encrypted using the cipher described in the previous part of the assignment with a period of `period`. The argument `key_square` provides the 9×9 key square required by the algorithm. For your convenience, `key_square` is null-terminated. As with the `bifid_encrypt` function, you are provided memory buffers `index_buffer` and `block_buffer`, which have the same expected sizes as in `bifid_encrypt`. The function writes the decrypted ciphertext into the `plaintext` buffer and null-terminates the string. The function returns the number of blocks (complete or partial blocks) created during the execution of the algorithm.

The function takes the following arguments, in this order:

- `plaintext`: The starting address of a memory buffer where the decrypted cipherext is written.
- `ciphertext`: The starting address of a null-terminated string to decrypt. The ciphertext is guaranteed to contain only those characters that appear in the key square.
- `key_square`: The 81-character, 2D array of uppercase letters, lowercase letters, digits and other ASCII characters used to decrypt the ciphertext. Characters are stored in row-major-order.
- `period`: The period of the cipher, i.e., how many pairs of indices are taken at a time and mapped to 2D coordinates. If `period` is negative or zero, the function returns `-1` and makes no changes to memory.
- `index_buffer`: A region of memory exactly `2 * strlen(plaintext))` bytes in size that can be used to temporarily store 8-bit unsigned integers. The address of this buffer is passed to the function at memory address `0($sp)`.
- `block_buffer`: A region of memory exactly `2 * period` bytes in size that can be used to temporarily store ASCII characters. The address of this buffer is passed to the function at memory address `4($sp)`.

Returns in `$v0`:

- The number of blocks of indices created by the algorithm, or `-1` on error when the period is negative or zero.

Additional requirements:

- The function must not write any changes to main memory except as specified.
- `bifid_decrypt` must call `index_of`.

Examples

In the examples below, the outermost set of double quotation marks that are typically used indicate a string have been deliberately omitted for sake of clarity.

Example #1:

```
ciphertext = S(vtjrB3drm,UNBn1!plm?\0
key_square = #8yszp,gv4\"huWA!w=lMRatXrEDGeBm5j$Z1Iq?Q*n;   (cont. next line)
             PTNV2iC9.H)k06o(f3@d7\'S:%OYK-xL/cUJ bF
period = 6
```

Expected `plaintext = Stony Brook University\0`

Expected return value: `4`

Example #2:

```
ciphertext = ZHFwS/xOoH-C*7$C\0"
key_square = 68*be/x!1uSvE0t?U;$l-ci(4nak=)2CW5\'ZOHPDAw   (cont. next line)
             f7qIXs#mjF\"VrLN@3p:BRJzdygGhTYMoQ9 %.,K
period = 2
```

Expected `plaintext = Computer Science\0`

Expected return value: `8`

Example #3:

```
ciphertext = DD$Vg5g$g$sDqx- 5oh-hTV\0
key_square = DQn\'v)t;iGI%5dOr*Rbu4zH@xcZ?:/M1XK,0P$9f=m   (cont. next line)
             eps(#VUg aBYCSj\"!-N.TlAkWLyh7FE3q2wo8J6
period = 200
```

Expected `plaintext = Dinner @ $50 per person\0`

Expected return value: `1`

Example #4:

```
ciphertext = DD$Vg5g$g$sDqx- 5oh-hTV\0
key_square = DQn'v)t;iGI%5dOr*Rbu4zH@xcZ?:/M1XK,0P$9f=m   (cont. next line)
             eps(#VUg aBYCSj"!-N.TlAkWLyh7FE3q2wo8J6"
period = -3
```

Expected `plaintext`: unchanged from the original value that was passed to the function.

Expected return value: `-1`

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 `proj2.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 `proj2.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 the steps again. We will grade only your last submission.