CSE 220: Systems Fundamentals I

Stony Brook University

Programming Project #2

Spring 2019

Assignment Due: Friday, March 15, 2019 by 11:59 pm

Updates to the Document:

3/12/2019: `encrypt` must be able to encrypt messages with both lowercase and uppercase letters, not just lowercase letters.

3/10/2019: `encrypt` may assume that the number of letters in `ciphertext` is greater than or equal to `ab_text_length`.

3/10/2019: `encrypt` does not need to call `count_letters`.

3/9/2019: Before making any function calls itself, `encrypt` should first check if `ab_text_length` is less than 10, and, if so, skip the call to `to_lowercase`. However, given that this was ambiguous in the original document, we will accept either behavior: calling or not calling `to_lowercase` under these circumstances.

3/4/2019: Corrected typo in Part IV to indicate that all characters of the plaintext must be encoded, not just lowercase letters. Corrected typo to indicate that the number of characters encoded in `ab_text` is what must be returned, not only the number of lowercase letters.

3/6/2019: `decode_plaintext` does not need to call `strlen`.

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 register 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 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 `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 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.
- 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.
- 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:

- 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. 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.
- 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).

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!

The Bacon Cipher

In this assignment you will implementing a simple encryption scheme known as Bacon's cipher. For each lowercase letter and for five other characters we assign a five-letter code as given in the below table:

Character	Encoding	Character	Encoding	Character	Encoding	Character	Encoding
a	AAAAA	i	ABAAA	q	BAAAA	y	BBAAA
b	AAAAB	j	ABAAB	r	BAAAB	z	BBAAB
c	AAABA	k	ABABA	s	BAABA	space	BBABA
d	AAABB	l	ABABB	t	BAABB	!	BBABB
e	AABAA	m	ABBAA	u	BABAA	'	BBBAA
f	AABAB	n	ABBAB	v	BABAB	,	BBBAB
g	AABBA	o	ABBBA	w	BABBA	.	BBBBA
h	AABBB	p	ABBBB	x	BABBB	eom	BBBBB

`"BBBAA"` is matched with a single quotation mark; `"BBBAB"` is matched with a comma. `"BBBBB"` is the code for "end-of-message", which is explained below.

Encryption

Encryption requires a plaintext message, which may consist of any uppercase letters, lowercase letters and the five other characters given in the table. The first step of encryption is to change all letters in the plaintext message to lowercase. Then, each character of the modified plaintext is mapped to its five-letter code.

For example, `"I'm.a.Seawolf!!"` is converted to `"i'm.a.seawolf!!"`. Then, each character is mapped to its five-letter code from the above table:

```
ABAAA BBBAA ABBAA BBBBA AAAAA BBBBA BAABA AABAA AAAAA BABBA ABBBA ABABB
  i     '     m     .     a     .     s     e     a     w     o     l
AABAB BBABB BBABB BBBBB
  f     !     !    eom
```

We refer to this encoding as "A/B text" of the encoded plaintext throughout this document. The end-of-message marker, `"BBBBB"`, is appended to the end of the A/B text, which is needed during the decryption process. Note that the number of characters in the A/B text is exactly $5n + 5$, where n is the length of the plaintext. Spaces between the five-letter codes are shown only for clarity and are not part of the actual A/B text.

The encryption algorithm also requires a string of text that may contain any combination of any printable characters. This text will be manipulated to become the ciphertext. Each letter of the ciphertext will encode either an A or a B from the A/B text, while other non-letter characters carry no information. An A from the A/B text is encrypted in the ciphertext as a lowercase letter, whereas a B from the A/B text is encrypted in the ciphertext as an uppercase letter. An example will help to clarify how this works.

Suppose we wanted to encode the A/B given above in the text "Python features a dynamic type system and automatic memory management. It supports multiple PROGRAMMING paradigms." On line 1 of each block below we see the ciphertext before the letter cases have been changed. On line 2 is the A/B text generated during the encoding process.

```
Python features a dynamic type system and automatic memory management.
ABAAAB BBAAABBA A BBBBAAA AAAB BBBABA ABA AABAAAAAA ABABBA ABBBAABABB

It supports multiple PROGRAMMING paradigms.
AA BABBBABB BBABBBBB BB
```

Final ciphertext:

```
pYthoN FEatuREs a DYNAmic typE SYStEm aNd auTomatic mEmORy mANAgeMeNT.
it SuPPOrTS MUlTIPLE PROGRAMMING paradigms.
```

Any letters from the ciphertext that follow the encoded end-of-message marker are left unmodified.

Decryption

Decryption is fundamentally the reverse process of encryption. Note that the original cases of the letters in the plaintext are lost during encryption. Therefore, the decryption process will generally not exactly reproduce the original plaintext message.

Suppose we have the following ciphertext:

```
Cse 220 AnD csE 320 FoRM A twO-CoURSe seQUeNce. 11001101 is a BInarY nUMBer.
81FE2D is A bAsE-16 NUMBer. Base conversion is FUN!
```

That ciphertext will be decoded into the following A/B text:

```
BAABABAABBABBBAABBABBBAAABBABAAAAABBAAABABBBAABBBAABABABBBBB
```

If we overlay the ciphertext with A/B text we will see why:

```
Cse 220 AnD csE 320 FoRM A twO-CoURSe seQUeNce. 11001101 is a BInarY nUMBer.
BAA     BAB AAB     BABB B AAB BABBBA AABBABAA           AA A BBAAAB ABBBAA

81FE2D is A bAsE-16 NUMBer.
  BB B  AAB ABABBBB B
```

The last few characters of the ciphertext don't need to be decoded because they lie after the end-of-message marker, `BBBBB`.

Now we take the A/B text and group it into five-letter groups that can be decoded:

```
BAABA BAABB ABBBA ABBAB BBAAA BBABA AAAAB BAAAB ABBBA ABBBA ABABA BBBBB
  S     T     O     N     Y   space   B     R     O     O     K
```

The decrypted message is always generated in uppercase letters, regardless of the letter cases in the original plaintext message.

Implementation

Before implementing the encryption and decryption algorithms we will need to write a few functions for working with strings.

Part I: Convert a String to Lowercase

```
int to_lowercase(string str)
```

This function takes a null-terminated string (possibly empty) and changes all of its uppercase letters to lowercase. All other characters in the string remain unchanged.

The function takes the following arguments, in this order:

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

Returns in `$v0`:

- The number of letters changed from uppercase to lowercase.

Additional requirements:

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

Examples:

Function Argument	Return Value	Updated `str`
`"Wolfie Seawolf!!!  2019??"`	2	`"wolfie seawolf!!!  2019??"`
`""`	0	`""`
`"programming"`	0	`"programming"`

Part II: 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
`"Wolfie Seawolf!!! 2019??"`	24
`"MIPS"`	4
`""`	0

Part III: Count the Number of Letters in a String

`int count_letters(string str)`

This function takes a null-terminated string (possibly empty) and returns the number of letters 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 number of letters in the string.

Additional requirements:

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

Examples:

Function Arguments	Return Value
`"Wolfie Seawolf!!! 2019??"`	13
`"220"`	0
`""`	0

Part IV: Encode Plaintext in "A/B" Format

```
(int, int) encode_plaintext(string plaintext, char[] ab_text,
                            int ab_text_length, char[] codes)
```

This function takes the null-terminated string `plaintext` (possibly empty) consisting of lowercase letters, spaces and punctuation marks, and encodes each character as its 5-letter uppercase Bacon code in `ab_text`. Only those punctuation marks and non-letter characters from the table on page 4 of this document will be present in the plaintext. After the last character of `plaintext` has been encoded and stored in `ab_text`, the function writes the string `"BBBBB"` into `ab_text` immediately after after the last encoded letter. `"BBBBB"` is called the "end-of-message marker." Because `ab_text` might not be long enough to accommodate the fully-encoded plaintext, the function encodes as many characters of the plaintext as possible, following them with the end-of-message code (`"BBBBB"`). The function does not null-terminate `ab_text`. In some cases, `ab_text` might not even be large enough to store the end-of-message marker. See the examples below.

The function takes the following arguments, in this order:

- `plaintext`: The starting address of the null-terminated string to encode.
- `ab_text`: The starting address of a memory buffer set aside to store the encoded plaintext.
- `ab_text_length`: The number of bytes in `ab_text`.
- `codes`: The starting address of the array described below.

The `codes` array will always have the following contents. This array is provided merely as a convenience to several of the functions for the assignment. Your functions are not required to use it.

```
# The codes for lowercase letters a through z are followed by codes
# for other characters.
.ascii "AAAAA"  # a
.ascii "AAAAB"  # b
.ascii "AAABA"  # c
# ... etc. ...
.ascii "BBAAA"  # y
.ascii "BBAAB"  # z
.ascii "BBABA"  # space
.ascii "BBABB"  # exclamation mark
.ascii "BBBAA"  # quotation mark
.ascii "BBBAB"  # comma
.ascii "BBBBA"  # period
.ascii "BBBBB"  # end-of-message marker
```

Returns in `$v0`:

- The number of characters from `plaintext` that were encoded as 5-letter codes in `ab_text`. This count does not include the end-of-message marker.

Returns in `$v1`:

- `1` if the entire plaintext was sucessfully encoded in `ab_text`; `0` if not all of the plaintext could be encoded in `ab_text`.

Additional requirements:

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

Examples:

The examples given below do not cover every possible edge case your function might encounter. Think carefully about special cases and, if it is not clear to you how the function should handle a special case, post a question on Piazza.

Case #1: `plaintext` is non-empty and `ab_text` is large enough to encode the entire plaintext message, including the end-of-message marker. The function completely encodes the plaintext and returns `N, 1`, where `N` is the number of characters in `plaintext`.

Function arguments:

```
plaintext = "java! not. a. fan."
ab_text = "**********************************************************
           **********************************"
ab_text_length = 98
```

Return values: `18, 1`

Updated `ab_text`: `"ABAABAAAAABABABAAAAABBABBBBABAABBABABBBABAABBBBBBABBABAAAAAA
BBBBABBABAAABABAAAAAABBABBBBBABBBBB***"`

Case #2: `plaintext` is non-empty and `ab_text` is not large enough to encode the entire plaintext message, but can accommodate at least one plaintext character's code, plus the end-of-message marker. The function encodes as much of the plaintext as possible, writes the end-of-message marker, and returns `N, 0`, where `N` is the number of characters in `plaintext` that were successfully encoded.

Function arguments:

```
plaintext = "we love mips..."
ab_text = "??????????????????????????????????????????????????????????"
ab_text_length = 63
```

Return values: `11, 0`

Updated ab_text: "BABBAAABAABBABAABABBABBBABABABAABAABBABAABBAAABAAAABBBBBB
BBB???"

Case #3: plaintext is non-empty and ab_text is not large enough to encode even a single character from the plaintext, but it does have enough room to store the end-of-message marker. The function writes only the end-of-message marker into ab_text and returns 0, 0.

Function arguments:

```
plaintext = "we love mips..."
ab_text = "*********"
ab_text_length = 9
```

Return values: 0, 0

Updated ab_text: "BBBBB****"

Case #4: plaintext is non-empty and ab_text is not large enough to encode even the end-of-message marker. The function writes no changes to ab_text and returns 0, 0.

Function arguments:

```
plaintext = "we love mips..."
ab_text = "@@@"
ab_text_length = 3
```

Return values: 0, 0

Updated ab_text: "@@@"

Case #5: plaintext is empty and ab_text is large enough to encode the end-of-message marker. The function writes the end-of-message marker to ab_text and returns 0, 1.

Function arguments:

```
plaintext = ""
ab_text = "!!!!!!!!!!!!!!!!"
ab_text_length = 16
```

Return values: 0, 1

Updated ab_text: "BBBBB!!!!!!!!!!"

Case #6: plaintext is empty and ab_text is not large enough to encode the end-of-message marker. The function writes no changes to ab_text and returns 0, 0.

Function arguments:

```
plaintext = ""
ab_text = "gg"
ab_text_length = 2
```

Return values: `0, 0`

Updated `ab_text`: `"gg"`

Part V: Encrypt a Message

```
(int, int) encrypt(string plaintext, string ciphertext, char[] ab_text,
                   int ab_text_length, char[] codes)
```

This function takes the null-terminated string `plaintext` (possibly empty) consisting of ~~lowercase~~ letters, spaces and punctuation marks, and encrypts the plaintext using Bacon's cipher as explained earlier in the assignment, storing the result in `ciphertext`. `encrypt` may assume that the number of letters in `ciphertext` is greater than or equal to `ab_text_length`.

The function takes the following arguments, in this order:

- `plaintext`: The starting address of the null-terminated string to encrypt.
- `ciphertext`: The starting address of the null-terminated string that will be modified to store the encrypted plaintext. The function may change the case of only those letters that must be changed to represent A/B letters taken from the `ab_text`. No other characters may be changed. For example, once `"BBBBB"` has been represented as five sequential uppercase letters in the ciphertext string, no characters after those five letters from the ciphertext may be modified. See the examples below for further explanation.
- `ab_text`: The starting address of a memory buffer created to store the encoded plaintext. You may not assume that `ab_text` is long enough to encode the entire plaintext. In such cases, only a portion of the plaintext will be encrypted.
- `ab_text_length`: The number of bytes in `ab_text`.
- `codes`: The starting address of the array of Bacon cipher codes described earlier in the assignment.

Returns in `$v0`:

- The number of letters of `ciphertext` that actually encode A/B code letters. See the examples below for further explanation.

Returns in `$v1`:

- `1` if the entire plaintext was successfully encrypted in `ciphertext`; `0` if not all of the plaintext could be encrypted because `ab_text` is too short.

Additional requirements:

- The function must not write any changes to main memory except as specified. As an example, this function

must not write any changes to `ab_text`; only `encode_plaintext` may write such changes.

- `encrypt` must call `to_lowercase` ~~`, count_letters`~~ and `encode_plaintext`.

Examples:

The examples given below do not cover every possible edge case your function might encounter. Think carefully about special cases and, if it is not clear to you how the function should handle a special case, post a question on Piazza.

Case #1: `plaintext` is non-empty and `ab_text` is large enough to encode the entire plaintext message, including the end-of-message marker. The function completely encrypts the plaintext and returns `5*(N+1), 1`, where `N` is the number of characters in `plaintext`.

Function arguments:

```
plaintext = "Stony Brook"
ciphertext = "CSE 220 and CSE 320 form a two-course sequence. 11001101 is a
              binary number. 81FE2D is a base-16 number. Base conversion is
              FUN!"
ab_text =  "************************************************************
            ****"
ab_text_length = 67
```

Return values: `60, 1`

Updated `plaintext`: `"stony brook"`

Updated `ciphertext`: `"Cse 220 AnD csE 320 FoRM A twO-CoURSe seQUeNce.  11001101 is a BInarY nUMBer. 81FE2D is A bAsE-16 NUMBer.  Base conversion is FUN!"`

Note that the substring `"E-16 NUMB"` encodes end-of-message marker, `"BBBBB"` and that the remainder of the initial `ciphertext` remains unchanged: `"er. Base conversion is FUN!"`.

Case #2: `plaintext` is non-empty and `ab_text` is not large enough to encode the entire plaintext message, but can accommodate at least one plaintext character's code, plus the end-of-message marker. The function encrypts as much of the plaintext as possible and returns `5*(N+1), 0`, where `N` is the number of characters in `plaintext` that were successfully encrypted.

Function arguments:

```
plaintext = "I'm.a.Seawolf!!"
ciphertext = "Python features a dynamic type system and automatic memory
              management. It supports multiple PROGRAMMING paradigms."
ab_text = "&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&&
           &&&&&&&&&&"
ab_text_length = 72
```

Return values: `70, 0`

Updated `plaintext`: `"i'm.a.seawolf!!"`

Updated `ciphertext`: `"pYthoN FEatuREs a DYNAmic typE SYStEm aNd auTomatic mEmORy mANAgeMeNT. it SuPPORTS multiple PROGRAMMING paradigms."`

Case #3: `plaintext` is non-empty and `ab_text` is not large enough to encode even a single character from the plaintext, but it does have enough room to store the end-of-message marker. The function encrypts only the end-of-message marker and returns `5, 0`.

Function arguments:

```
plaintext = "Wolfie Seawolf"
ciphertext = "Python features a dynamic type system and automatic memory
              management. It supports multiple programming paradigms."
ab_text = "******"
ab_text_length = 6
```

Return values: `5, 0`

Updated `plaintext`: `"wolfie seawolf"` or `"Wolfie Seawolf"`

Updated `ciphertext`: `"PYTHOn features a dynamic type system and automatic memory management. It supports multiple programming paradigms."`

Case #4: `plaintext` is non-empty and `ab_text` is not large enough to encode even the end-of-message marker. The function writes no changes to `ab_text` or `ciphertext` and returns `0, 0`.

Function arguments:

```
plaintext = "Wolfie Seawolf"
ciphertext = "Python features a dynamic type system and automatic memory
              management. It supports multiple programming paradigms."
ab_text = "%%"
ab_text_length = 2
```

Return values: `0, 0`

Updated `plaintext`: `"Wolfie Seawolf"` or `"wolfie seawolf"`

Updated `ciphertext`: `"Python features a dynamic type system and automatic memory management. It supports multiple programming paradigms."`

Case #5: `plaintext` is empty and `ab_text` is large enough to encode the end-of-message marker. The function call to `encode_plaintext` writes the end-of-message marker to `ab_text` and then `encrypt` returns `5, 1`.

Function arguments:

```
plaintext = ""
ciphertext = "The Scholars program is a vibrant community of high-achieving
              students at Stony Brook."
ab_text = "!!!!!!!!!!!!!!!!"
ab_text_length = 16
```

Return values: `5, 1`

Updated `plaintext`: `""`

Updated `ciphertext`: `"THE SCholars program is a vibrant community of high-achieving students at Stony Brook."`

Case #6: `plaintext` is empty and `ab_text` is not large enough to encode the end-of-message marker. No changes are written `ab_text` or `plaintext`. The function returns `0, 0`.

Function arguments:

```
plaintext = ""
ciphertext = "The Scholars program is a vibrant community of high-achieving
              students at Stony Brook."
ab_text = "!!!!"
ab_text_length = 4
```

Return values: `0, 0`

Updated `plaintext`: `""`

Updated `ciphertext`: `"The Scholars program is a vibrant community of high-achieving students at Stony Brook."`

Part VI: Decode Ciphertext into an "A/B" String

```
int decode_ciphertext(string ciphertext, char[] ab_text, int ab_text_length,
                      char[] codes)
```

This function takes a string of ciphertext and decodes it into a string of A and B characters. In sequential manner, starting at the leftmost character of `ciphertext`, each lowercase letter is encoded as an `A` character in `ab_text`, and each uppercase letter is encoded as a `B` character in `ab_text`. Non-letter characters in `ciphertext` are simply ignored. You may assume that the end-of-message marker is encoded somewhere in the ciphertext. The function does not null-terminate the `ab_text` string.

If the number of letters in `ciphertext` is greater than the length of `ab_text`, the function returns `-1` and makes no changes to memory.

The function takes the following arguments, in this order:

- `ciphertext`: The starting address of the null-terminated ciphertext string to decode as explained above.

- `ab_text`: The starting address of a memory buffer created to store the decoded ciphertext.
- `ab_text_length`: The size of the `ab_text` in bytes.
- `codes`: The starting address of the array of Bacon cipher codes described earlier in the assignment.

Returns in `$v0`:

- The number of characters written to `ab_text` by the function, or `-1` on error as described above.

Additional requirements:

- The function must not write any changes to main memory except as specified.
- `decode_ciphertext` must call ~~`strlen` and~~ `count_letters`.

Examples:

The examples given below do not cover every possible edge case your function might encounter. Think carefully about special cases and, if it is not clear to you how the function should handle a special case, post a question on Piazza.

Case #1: `ab_text` is exactly the right length to store the decoded ciphertext.

Function arguments:

```
ciphertext = "AbcDefgHijkLMnOpqrSTUVwXyzaBCDefGhijKlMNoPQRSTUvWXYZABC"
ab_text =    "////////////////////////////////////////////////////"
ab_text_length = 55
```

Return value: `55`

Updated `ab_text`: `"BAABAAABAAABBABAAABBBBABAAABBBAABAAABABBABBBBBBABBBBBBB"`

Case #2: `ab_text` has bytes leftover after storing the decoded ciphertext

Function arguments:

```
ciphertext = "sHARDPLate Is aN AncIENt AND maGIcAL TyPe OF fUlL-BODy ARMOr
              FOUNd ON ROsHAr. It iS infUseD wITh StoRMLigHt AnD GRANts
              great power."
ab_text =    "**********************************************************
              **************************************************"
ab_text_length = 114
```

Return value: `95`

Updated `ab_text`: `"ABBBBBBAAABAABBAABBBABBBAABBABBBABABBABABBBBABBBBABBBBABBBBAB
BABAABAAABAABABBABAABBBAABABABBBBB**********"`

Case #3: `ciphertext` contains only the end-of-message marker.

Function arguments:

```
ciphertext = "2019 GO STONY BROOK!"
ab_text =    "+++++++++++++++"
ab_text_length = 15
```

Return value: `5`

Updated `ab_text`: `"BBBBB++++++++++"`

Part VII: Decrypt a Message

```
int decrypt(string ciphertext, string plaintext, char[] ab_text,
            int ab_text_length, char[] codes)
```

This function takes a ciphertext string as its first argument, calls `decode_ciphertext` to transform the ciphertext into an A/B string, and then finally decrypts the A/B string, storing the decrypted plaintext in the given `plaintext` buffer. The function null-terminates the plaintext before returning.

If the call to `decode_ciphertext` returns `-1`, `decrypt` returns `-1` and makes no changes to memory.

The function takes the following arguments, in this order:

- `ciphertext`: The starting address of the null-terminated ciphertext string to decrypt.
- `plaintext`: The starting address of a memory buffer to store the plaintext. The buffer is guaranteed to be large enough to store the decrypted message and a null-terminator.
- `ab_text`: The starting address of a memory buffer created to store the decoded ciphertext.
- `ab_text_length`: The size of the `ab_text` in bytes.
- `codes`: The starting address of the array of Bacon cipher codes described earlier in the assignment. Note that this argument will be passed to `decrypt` on the stack at memory address `0($sp)`.

Returns in `$v0`:

- The number of characters in the decrypted message, not including the null-terminator written by the function, or `-1` on error as described above.

Additional requirements:

- The function must not write any changes to main memory except as specified. As an example, this function must not write any changes to `ab_text`; only `decode_ciphertext` may write such changes.
- `decrypt` must call `decode_ciphertext`.

Examples:

The examples given below do not cover every possible edge case your function might encounter. Think carefully about special cases and, if it is not clear to you how the function should handle a special case, post a question on Piazza.

Case #1: General case. The ciphertext contains at least one printable character.

Function arguments:

```
ciphertext = "Cse 220 AnD csE 320 FoRM A twO-CoURSe seQUeNce. 11001101 is a
              BInarY nUMBer. 81FE2D is A bAsE-16 NUMBer. Base conversion is
              FUN!"
plaintext =  "@@@@@@@@@@@@@@@@@@@@@@@@@"
ab_text =    "%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
              %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
              %%%%%%"
ab_text_length = 81
```

Return value: `11`

Updated `plaintext`: `STONY BROOK\0@@@@@@@@@@@@@`

Updated `ab_text`: `BAABABAABBABBBAABBABBBAAABBABAAAAABBAAABABBBAABBBAABABABBBBB%%`
`%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%`

Case #2: `ciphertext` contains only the end-of-message marker.

Function arguments:

```
ciphertext = "2019 GO STONY BROOK!"
plaintext  = "&&&&&&&&&&&&&&"
ab_text =    "++++++++++++++++"
ab_text_length = 16
```

Return value: `0`

Updated `plaintext`: `\0&&&&&&&&&&&&&`

Updated `ab_text`: `"BBBBB+++++++++++"`

Case #3: `ciphertext` contains more letters than can be decoded into `ab_text`

Function arguments:

```
ciphertext = "Cse 220 AnD csE 320 FoRM A twO-CoURSe seQUeNce. 11001101 is a
              BInarY nUMBer."
plaintext  = "&&&&&&&&&&&&&"
ab_text =    "????????????????????????????????????"
ab_text_length = 36
```

Return value: `-1`

Updated `plaintext`: `&&&&&&&&&&&&`

Updated `ab_text`: `"????????????????????????????????????"`

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.