CSE 220: Systems Fundamentals I

Stony Brook University

Programming Assignment #2

Fall 2020

Assignment Due: Sunday, October 11, 2020 by 11:59 pm EDT

Updates to this Document

- 10/4/2020: Step 10 of the algorithm in Part 10 had a typo. It has now been fixed and highlighted in red.
- 9/26/2020: Updated examples in Part 6.
- 9/26/2020: You can now copy text from the document. DO NOT DOWNLOAD THE FILE or you will likely miss future updates to the document.

Learning Outcomes

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

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

Getting Started

Visit the course website and download the files hwk2.zip and MarsFall2020.jar. Fill in the following information at the top of hwk2.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 hwk2.asm you will find several function stubs that consist simply of `jr $ra` instructions. Your job
in this assignment is to 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 hwk2.asm. Helper functions will not necessarily be graded, but might be spot-checked to ensure you are following the MIPS function calling conventions.

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 hwk2.asm file. A submission that contains a `.data` section will probably result in a score of zero.

Important Information about CSE 220 Programming Projects

- Read the entire project description document 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 the course website. 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 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 the assignment if you do this.
- Submit your final .asm file to the course website by the due date and time. Late work will be penalized as described in the course syllabus. 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 few exceptions, all aspects of your programming assignments 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 200,000 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.) will 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 implemented 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 append the contents of your hwk2.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 with your hwk2.asm file – we will delete them. Also please note that these testing main files will not be used in grading.

Again, any modifications to the main files will not be graded. You will submit only your hwk2.asm for grading. Make sure that all code required for implementing your functions is included in the hwk2.asm file. To make sure that your code is self-contained, try assembling your hwk2.asm file by itself in MARS. If you get any errors (such as a missing label), this means that your hwk2.asm file is attempting to reference labels in the main file, which will not have the same names during grading!

A Final 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!

Preliminaries

For this assignment you will be implementing several algorithms for working with ASCII strings, as well as encryption and decryption algorithms for a variant of the homophonic substitution cipher. In this cipher, some of the letters from the *plaintext* (the input message) can be replaced with one of several different possible letters to generate the *ciphertext* (the encrypted output message). This is in contrast with a traditional substitution cipher, wherein there is only one option to substitute for each plaintext letter. If you are unfamiliar with substitution ciphers, we recommend you briefly review the Wikipedia article on the topic.

As a concrete example, suppose the *plaintext alphabet* is:

```
aaaaaaabcdeeeeeeeeefghhiiiijklmnnnnnooooopqrrrsttttttttuvwxyz
```

and the *ciphertext alphabet* is:

```
WhatsSewolfImAOF20BUCVD19ZEdinbcgjkpqruvxyzGHJKLMNPQRTXY345678
```

The above alphabets indicate that ‘a’ can be replaced any of the characters “`WhatsSe`” in the ciphertext; ‘n’ can be replaced with any of the characters “`cgjkp`”, and so on;

In our cipher, encryption and decryption require two secret pieces of information: a *keyphrase*, which is intended to be an easily-remembered phrase or sentence, and a longer *corpus*, which is at least several sentences of text from a document that can be easily obtained (e.g., from a library, the Web, etc.) In a homophonic substitution cipher, the idea is that a letter which occurs frequently in English text can be substituted with any of several possible characters, not just the same character every time. This makes it harder to crack the code. Less-frequently occurring letters might be substituted with only a handful of letters in the ciphertext, or perhaps only even one. The corpus text is used in the frequency analysis to decide how many different possible substitutions could be performed for each plaintext letter. The keyphrase determines the order of the symbols in the ciphertext alphabet, which contains the characters which will be substituted for the plaintext letters. All of this will become clearer as we work through an example.

Encryption Phase 1: Create the Ciphertext Alphabet using the Keyphrase

Our ciphertext alphabet contains 62 symbols that will be used to replace the plaintext letters: 26 lowercase letters, 26 uppercase letters and 10 digit characters provide the 62 symbols.

1. Initialize an empty ciphertext alphabet consisting of 62 bytes.
2. Draw letters and digit characters one at a time from the keyphrase, left-to-right, possibly adding each character to the ciphertext alphabet. Ignore spaces and punctuation marks.
    a. If a drawn character has not already been added to the ciphertext alphabet, then add it to the end.
    b. Otherwise, skip that character and move on to the next character of the keyphrase.
3. After drawing all alphanumeric characters from the keyphrase, determine which lowercase letters do not appear in the ciphertext alphabet. Add these missing letters in alphabetical order to the ciphertext alphabet.
4. Repeat step 3, but for uppercase letters.
5. Repeat step 3. but for digit characters.

As an example, suppose the keyphrase is:

```
What's a Seawolf? I'm a SeAwOlF! Fall 2020 SBU: COVID-19 Zoom Edition
```

After step 2, the (incomplete) ciphertext alphabet will consist of these characters:

```
WhatsSewolfImAOF20BUCVD19ZEdin
```

After step 3, the missing lowercase letters have been appended to yield:

```
WhatsSewolfImAOF20BUCVD19ZEdinbcgjkpqruvxyz
```

After step 4, the missing uppercase letters have been appended to yield:

```
WhatsSewolfImAOF20BUCVD19ZEdinbcgjkpqruvxyzGHJKLMNPQRTXY
```

After step 5, the missing digit characters have been appended to yield the final ciphertext alphabet:

```
WhatsSewolfImAOF20BUCVD19ZEdinbcgjkpqruvxyzGHJKLMNPQRTXY345678
```

Encryption Phase 2: Compute the Frequency of Letters in the Corpus and Assign Substitutions

In this phase, we first count the number of times each letter occurs in the corpus, ignoring case. Then, we sort the letters into descending order by count. In other words, the most frequently occurring letters are placed near the front of the sorted alphabet. If two letters have the same frequency, the one with the lower ASCII value is placed in front of the other.

Consider the following text, used as the corpus:

> Four score and seven years ago our fathers brought forth on this continent a new nation, conceived in Liberty, and dedicated to the proposition that all men are created equal. Now we are engaged in a great civil war, testing whether that nation or any nation so conceived and so dedicated, can long endure. We are met on a great battle-field of that war. We have come to dedicate a portion of that field, as a final resting place for those who here gave their lives that that nation might live. It is altogether fitting and proper that we should do this.

This text generates the following case-insensitive counts:

```
a 46   b 3   c 14   d 21   e 57   f 9   g 13   h 23   i 32   j 0   k
0
l 14   m 4   n 35   o 37   p  6   q 1   r 28   s 16   t 52   u 6   v
8
w  8   x 0   y  3   z  0
```

Sorting the letters according the criteria explained above yields the string:

```
etaonirhdsclgfvwpumbyqjkxz
```

Next, to generate the plaintext alphabet, we duplicate the most frequently occurring letter eight times, the second most frequently occurring letter seven times, …, and the eighth most frequently occurring letter once. (Note that 8+7+6+5+4+3+2+1 = 36 and that 26+36 = 62.) For our example, we obtain:

```
aaaaaaabcdeeeeeeeeefghhiiiijklmnnnnnooooopqrrrsttttttttuvwxyz
```

Note that ‘e’ is repeated eight times, the ‘t’ is repeated seven times, the ‘a’ is repeated six times, etc. The ‘h’, which is the eighth most frequently occurring letter, is repeated only once. All other letters appear only once each. Note that the plaintext alphabet contains only lowercase letters. This means that any uppercase letters in the input message must be changed to lowercase during the encryption process.

Encryption Phase 3: Perform Substitutions to Generate the Ciphertext

Now we lay the plaintext alphabet on top of the ciphertext alphabet:

```
aaaaaaabcdeeeeeeeeefghhiiiijklmnnnnnooooopqrrrsttttttttuvwxyz
WhatsSewolfImAOF20BUCVD19ZEdinbcgjkpqruvxyzGHJKLMNPQRTXY345678
```

We see that, in theory, ‘a’ can be replaced with any of the seven characters from the string “WhatsSe”; ‘b’ can be replaced only with ‘w’; ‘c’ can be replaced only with ‘o’; ‘d’ can be replaced only with ‘l’; ‘e’ can be replaced with any of the nine characters from “fImAOF20B”, and so on. To make this selection process deterministic (non-random), we will use the index of a character from the plaintext *modulo* the number of instances of a character to choose the substituted character from the ciphertext alphabet. For instance, suppose there is a letter ‘e’ at index 12 of the plaintext. Note that there are nine instances of ‘e’ in the ciphertext alphabet. 12 mod 9 = 3, so we take the character at index 3 of the string “fImAOF20B”, which is ‘A’. Note that in all computations we assume that indexes strings and substrings are 0-based. Namely, the leftmost character of the plaintext is 0; the characters of the plaintext alphabet and ciphertext alphabet are indexed 0 through 61 (inclusive); substrings drawn from the ciphertext alphabet are indexed starting from 0, etc. Finally, note that only lowercase letters from the plaintext are encrypted; all other characters (punctuation marks, spaces, etc.) are merely copied to the ciphertext unencrypted. We will assume for this assignment that the plaintext message will never contain digits.

Part 1: Compute the Length of a Null-terminated String

```
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 null-terminator is guaranteed to be present and is not counted in the length.

The function takes one argument:

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

Returns in $v0:

- The number of characters in the string, not including the null-terminator.

Additional requirements:

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

Examples:

Function Argument	Return Value
`"Wolfie Seawolf!!! 2020??"`	`24`
`"MIPS"`	`4`
`""` (empty string, containing only \0)	`0`

Part 2: Find the Index of a Character in a Null-terminated String

`int index_of(string str, char ch, int start_index)`

The function returns the index of the first instance of printable character `ch` in the null-terminated string `str`, starting the search at index `start_index` and moving to the right (towards higher indexes). If the character is not present in the string or if `start_index` is not a valid index for `str`, 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 printable character to search for
- `start_index`: the index at which the search begins

Returns in $v0:

- The index of the leftmost occurrence of `ch` in `str` found at index `start_index` or to the right of `start_index`; or -1 if `ch` is not found during the search or if `start_index` is not a valid index for `str`.

Additional requirements:

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

Examples:

Function Arguments	Return Value
`"CSE 220 COVID-19 Edition", 'V', 3`	`10`

`"CSE 220 COVID-19 Edition", 'V', 13`	`-1`
`"CSE 220 COVID-19 Edition", 'n', 5`	`23`
`"CSE 220 COVID-19 Edition", 'n', -5`	`-1`
`"", 'z', 0`	`-1`

Part 3: Convert a Null-terminated 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 one argument:

- `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 for `str`.

Examples:

Function Arguments	**Return Value**
`"Stony Brook University"`	`3`
`"UNIVERSITY"`	`10`
`"2020-2021"`	`0`
`""`	`0`

Part 4: Generate the Ciphertext Alphabet from a Keyphrase

```
int generate_ciphertext_alphabet(string ciphertext_alphabet,
                                 string keyphrase)
```

This function generates the ciphertext alphabet as described in Encryption Phase 1 in the Preliminaries section of the document above. The function must null-terminate `ciphertext_alphabet` by writing a null-terminator at `ciphertext_alphabet[62]`.

The function takes the following arguments, in this order:

- `ciphertext_alphabet`: an uninitialized buffer of 63 contiguous bytes of main memory where the function writes the ciphertext alphabet.
- `keyphrase`: a non-empty, null-terminated string to serve as the keyphrase

Returns in $v0:

- The number of unique, case-sensitive alphanumerical characters drawn from `keyphrase`.

Additional requirements:

- The function must not write any changes to main memory except for `ciphertext_alphabet`. For example, the string `keyphrase` must remain unchanged.

Example #1:

```
keyphrase = "Stony Brook University"
```
Resulting `ciphertext_alphabet`:
```
          "StonyBrkUivesabcdfghjlmpquwxzACDEFGHIJKLMNOPQRTVWXYZ0123456789
     "
```

Returns in $v0: `13`

Example #2:

```
keyphrase = "Monday, September 21st, 2020 4:39 PM EDT"
```
Resulting `ciphertext_alphabet`:
```
          "MondaySeptmbr21s0439PEDTcfghijklquvwxzABCFGHIJKLNOQRUVWXYZ5678
     "
```

Returns in $v0: `24`

Example #3:

```
keyphrase = "suPeRcalIfrAgiListICexPiaLIdoCIOus"
```
Resulting `ciphertext_alphabet`:

```
"suPeRcalIfrAgiLtCxdoObhjkmnpqvwyzBDEFGHJKMNQSTUVWXYZ0123456789"
```

Returns in $v0: `21`

Part 5: Count the Occurrences of Each Lowercase Letter in a String

```
int count_lowercase_letters(int[] counts, string message)
```

This function counts the number of times each lowercase letter occurs in `message`, storing those counts in `counts`. Specifically, `counts[0]` stores the number of instances of 'a' in `message`, `counts[1]` the number of instances of 'b' in `message`, etc. Note that each element in `counts` is a 4-byte integer. `counts` is not initialized with zeros when the function is called.

The function takes the following arguments, in this order:

- `counts`: an uninitialized buffer of 26 contiguous words of main memory where the function writes the counts of the lowercase letters.
- `message`: a null-terminated string (possibly empty) that could consist of any characters with ASCII codes 32 through 126, inclusive

Returns in $v0:

- The total number of lowercase letters in `message`.

Additional requirements:

- The function must not write any changes to main memory except for `counts`. For example, the string `message` must remain unchanged.

Example #1:

```
message = "The specialization in artificial intelligence and data science
emphasizes modern approaches for building intelligent systems using
machine learning."
```

Resulting `counts`: `12 1 7 4 16 2 5 4 18 0 0 8 4 14 4 4 0 5 9 7 2 0 0 0 1 2`

which means:

```
a:12   b: 1   c: 7   d: 4   e:16   f: 2   g: 5   h: 4   i:18   j: 0   k: 0
l: 8   m: 4   n:14   o: 4   p: 4   q: 0   r: 5   s: 9   t: 7   u: 2   v: 0
w: 0   x: 0   y: 1   z: 2
```

Return value in $v0: `129`

Example #2:

```
message = "We can only see a short distance ahead, but we can see plenty
there that needs to be done. -Alan Turing"
```

Resulting `counts`: `8 2 3 4 15 0 1 4 2 0 0 3 0 9 4 1 0 3 5 8 2 0 1 0 2 0`

which means:

```
a: 8   b: 2   c: 3   d: 4   e:15   f: 0   g: 1   h: 4   i: 2   j: 0   k: 0
l: 3   m: 0   n: 9   o: 4   p: 1   q: 0   r: 3   s: 5   t: 8   u: 2   v: 0
w: 1   x: 0   y: 2   z: 0
```

Return value in $v0: `77`

Part 6: Sort the Letters of the Alphabet by Frequency

```
void sort_alphabet_by_count(string sorted_alphabet, int[] counts)
```

This function's purpose is to sort the lowercase letters of the Latin alphabet using the numbers given in the `counts` array as the sorting key, storing that sorted alphabet in `sorted_alphabet`. The buffer `sorted_alphabet` is guaranteed to be at least 27 bytes in size. The 26-word `counts` array provides a non-negative integer count associated with each lowercase letter. Specifically, `counts[0]` is the count for 'a', `counts[1]` is the count for 'b', etc. The function sorts the contents of `sorted_alphabet` so that the letter with highest count is at index 0, the letter with second-highest count is at index 1, and so. When two or more letters have the same count, the letter with smallest ASCII value is placed before the others in `sorted_alphabet`, the letter with second-smallest ASCII value immediately follows the letter with smallest ASCII value, etc. The function null-terminates `sorted_alphabet` by writing a null-terminator at `sorted_alphabet[26]`.

The function takes the following arguments, in this order:

- `sorted_alphabet`: an uninitialized buffer of 27 contiguous bytes of main memory.
- `counts`: an array of 26 non-negative counts. Each integer is four bytes. Note that this array is not necessarily the output of the `count_lowercase_letters` function, so do not assume that when coding this function.

Additional requirements:

- The function must not write any changes to main memory except for `sorted_alphabet` and `counts`.

Example #1:

```
counts = 28 13 24 2 28 19 12 2 0 10 23 14 3 28 1 2 21 4 4 25 29 0 9 29 13
18
```
Resulting `sorted_alphabet`: `"uxaentckqfzlbygjwrsmdhpoiv"`

Example #2:

```
counts = 21 17 20 25 21 19 28 26 15 16 21 13 11 16 1 27 24 20 5 23 26 2 29
15 21 8
```
Resulting `sorted_alphabet`: `"wgphudqtaekycrfbjnixlmzsvo"`

Example #3:

```
counts = 23 26 29 1 20 9 15 30 24 20 23 7 17 15 5 4 17 14 12 24 14 1 0 4
14 6
```
Resulting `sorted_alphabet`: `"hcbitakejmqgnruysflzopxdvw"`

Part 7: Generate the Plaintext Alphabet from a Sorted Alphabet

```
void generate_plaintext_alphabet(string plaintext_alphabet,
                                 string sorted_alphabet)
```

This function generates the 62-character plaintext alphabet from a sorted lowercase Latin alphabet using the algorithm described in Encryption Phase 2 under Preliminaries earlier in this document. The plaintext alphabet is written into the uninitialized `plaintext_alphabet` buffer, which is guaranteed to be at least 63 bytes in size. The function null-terminates `plaintext_alphabet` by writing a null-terminator at `plaintext_alphabet[62]`. Briefly, the plaintext alphabet contains the lowercase letters of the Latin alphabet in alphabetical order, except that `sorted_alphabet[0]` is repeated 8 times in `plaintext_alphabet`, `sorted_alphabet[1]` is repeated 7 times in `plaintext_alphabet`, etc., all the way to `sorted_alphabet[7]`, which is repeated once in `plaintext_alphabet`. The remaining 18 letters of `sorted_alphabet` each appears once in `plaintext_alphabet`.

The function takes the following arguments, in this order:

- `plaintext_alphabet`: an uninitialized buffer of 63 contiguous bytes of main memory where the function writes the 62-character plaintext alphabet, ending with a null-terminator
- `sorted_alphabet`: a null-terminated string containing the 26 lowercase letters of the Latin alphabet, in any order

Additional requirements:

- The function must not write any changes to main memory except for `plaintext_alphabet` and `sorted_alphabet`.

Example #1:

```
sorted_alphabet = "egljhotupvfsxawqkrmzdyncib"
```
Resulting `plaintext_alphabet`:
```
           "abcdeeeeeeeeefgggggggghhhhhijjjjjjjkllllllmnoooopqrstttuuvwxyz
     "
```

Example #2:

```
sorted_alphabet = "eznovrqbdatjghlwmskyipcxfu"
```
Resulting `plaintext_alphabet`:
```
           "abbcdeeeeeeeeefghijklmnnnnnnnnooooopqqqrrrrstuvvvvvwxyzzzzzzzzz
     "
```

Example #3:

```
sorted_alphabet = "jmhoxqzgityudwsecvfalnkrbp"
```

Resulting `plaintext_alphabet`:

```
           "abcdefgghhhhhhhhijjjjjjjjjklmmmmmmmmnooooopqqqqrstuvwxxxxxyzzz
     "
```

Part 8: Return the Ciphertext Character Substituted for a Letter from the Plaintext

```
int encrypt_letter(char plaintext_letter, int letter_index,
                   string plaintext_alphabet, string ciphertext_alphabet)
```

This function computes and returns the substitution of a plaintext letter for the ciphertext, given the plaintext letter itself, that letter's index in the plaintext message, the plaintext alphabet as generated by `generate_plaintext_alphabet` and the ciphertext alphabet as generated by `generate_ciphertext_alphabet`. The substitution process is described in Encryption Phase 3 in the Preliminaries section of the document, above. Briefly, suppose the `plaintext_letter` (located at index `letter_index` of some plaintext) appears at indexes `i` through `i+k` of `plaintext_alphabet`. The function will return the character at location `ciphertext_alphabet[i+(letter_index mod (k+1))]`. Note that it is this function's responsibility to determine the value of `k` for a given letter in `plaintext_alphabet`.

The function takes the following arguments, in this order:

- `plaintext_letter`: a lowercase letter
- `letter_index`: a non-negative integer
- `plaintext_alphabet`: the 62-character plaintext alphabet required to encrypt a plaintext message
- `ciphertext_alphabet`: the 62-character ciphertext alphabet required to encrypt a plaintext message

Returns in $v0:

- The encrypted letter or -1 if `plaintext_letter` is not a lowercase letter.

Additional requirements:

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

Example #1:

```
plaintext_letter = 'u'
letter_index = 3
plaintext_alphabet =
           "abbbbbbbbcdeffffffgggghiiijkklmnopqrstuuuuuvvvvvvwxxxxxxxxxyz
     "
ciphertext_alphabet =
           "StonyBrkUivesNwYadfAmcbghjlpquxzCDEFGHIJKLMOPQRTVWXZ0123456789
     "
```

Return value in $v0: `77` (ASCII code for `'M'`)

Example #2:

```
plaintext_letter = 'p'
letter_index = 46
plaintext_alphabet =
          "abcdeeefghijkkkkkkllllllmmnopppppppppqrstuvvvvvwxyyyyzzzzzzzzz
     "
ciphertext_alphabet =
          "StonyBrkUivesNwYadfAmcbghjlpquxzCDEFGHIJKLMOPQRTVWXZ0123456789
     "
```

Return value in $v0: `70` (ASCII code for `'F'`)

Example #3:

```
plaintext_letter = 'x'
letter_index = 37
plaintext_alphabet =
          "abccccccdeeeeeeeeeffffffffgghijkkklmnoppppqrstuvwxxxxxxyzzzzzzzz
     "
ciphertext_alphabet =
          "StonyBrkUivesNwYadfAmcbghjlpquxzCDEFGHIJKLMOPQRTVWXZ0123456789
     "
```

Return value in $v0: `87` (ASCII code for `'W'`)

Example #4:

```
plaintext_letter = 'n'
letter_index = 15
plaintext_alphabet =
          "abccccccccddddddefghiiiiiiijjjjjjjjjklmmnopqrstuvwwwwwwwxyyyyzzz
     "
ciphertext_alphabet =
          "StonyBrkUivesNwYadfAmcbghjlpquxzCDEFGHIJKLMOPQRTVWXZ0123456789
     "
```

Return value in $v0: `73` (ASCII code for `'I'`)

Part 9: Encrypt a Plaintext Message using a Homophonic Substitution Cipher

```
int, int encrypt(string ciphertext, string plaintext, string keyphrase,
                 string corpus)
```

This function encrypts the given plaintext message using the homophonic substitution cipher described earlier in the document, storing the resulting ciphertext in `ciphertext`. Assume that the plaintext contains only letters, spaces and punctuation marks; no digits will be present in the plaintext. All arguments are assumed to be valid. The buffer for the ciphertext is guaranteed to be large enough to store the null-terminated `ciphertext`. Note that the function returns two values.

The function takes the following arguments, in this order:

- `ciphertext`: an uninitialized buffer to store the encrypted plaintext
- `plaintext`: a null-terminated string to be encrypted using the homophonic substitution cipher
- `keyphrase`: a null-terminated keyphrase string used during the encryption process
- `corpus`: a null-terminated corpus string used during the encryption process

The function implements the following algorithm:

1. Call `to_lowercase` on both `plaintext` and `corpus`.
2. Allocate at least 26 words' worth of memory on the stack to temporarily store the `counts` array needed in the following step. (How? Simply subtract however many bytes you want to allocate from $sp. Make sure the number of bytes you allocate is a multiple of 4.) Be sure to deallocate this memory later.
3. Call `count_lowercase_letters(counts, corpus)`.
4. Allocate at least 27 bytes of memory on the stack to store the `lowercase_letters` string needed in the following step. Be sure to deallocate this memory later.
5. Call `sort_alphabet_by_count(lowercase_letters, counts)`.
6. Allocate at least 63 bytes of memory on the stack to temporarily store the `plaintext_alphabet` string needed in the following step. Be sure to deallocate this memory later.
7. Call `generate_plaintext_alphabet(plaintext_alphabet, lowercase_letters)`.
8. Allocate at least 63 bytes of memory on the stack to temporarily store the `ciphertext_alphabet` string needed in the following step. Be sure to deallocate this memory later.
9. Call `generate_ciphertext_alphabet(ciphertext_alphabet, keyphrase)`.
10. In a loop, call `encrypt_letter` to encrypt each lowercase letter of `plaintext` and write the return value into `ciphertext`. Each non-lowercase letter of `plaintext` should not be passed as an argument to `encrypt_letter`, but should instead simply be copied to `ciphertext`.
11. Null-terminate the `ciphertext` string.

Returns in $v0:

- The number of lowercase letters that were encrypted during the encryption process.

Returns in $v1:

- The number of characters from the plaintext that were not encrypted and simply copied to `ciphertext`.

Additional requirements:

- The function must not write any changes to main memory except as needed.
- The function must call the functions `to_lowercase` (twice), `count_lowercase_letters`, `sort_alphabet_by_count`, `generate_ciphertext_alphabet`, `generate_plaintext_alphabet`, and `encrypt_letter`.

Example #1:

```
plaintext = "Never trust a computer you can't throw out a window. -Steve
Wozniak"

keyphrase = "I'll have you know that I stubbed my toe last week and only
cried for 20 minutes."

corpus = "When in the Course of human events, it becomes necessary for one
people to dissolve the political bands which have connected them with
another, and to assume among the powers of the earth, the separate and
equal station to which the Laws of Nature and of Nature's God entitle
them, a decent respect to the opinions of mankind requires that they
should declare the causes which impel them to the separation."
```

Resulting `ciphertext`: `"Dk5wQ 1Q4RW h yLCO4XnQ 8H4 yaE'3 VpQH6 L4V a 6qGoJ6. -R3n5t 6J9GqIA"`

Return value in $v0: `53`

Return value in $v1: `14`

Example #2:

```
plaintext = "The trouble with having an open mind, of course, is that
people will insist on coming along and trying to put things in it. -Terry
Pratchett"

keyphrase = "What's the difference between ignorance and apathy? I don't
know and I don't care."

corpus = "Call me Ishmael. Some years ago - never mind how long precisely
- having little or no money in my purse, and nothing particular to
interest me on shore, I thought I would sail about a little and see the
watery part of the world. It is a way I have of driving off the spleen and
regulating the circulation."
```

Resulting `ciphertext`: `"Xjc 1UN4dDn 6xXj jW5vJk WJ ORcK GmKf, PI iO4TVn, mV 0jW3 RwQREy 6mDE xKVlV1 NJ iMGqLk aDPJk aJf 2U8uHk 2Q R40 2jlJkV xK lZ. -3rUT8 RThYijc23"`

Return value in $v0: `111`

Return value in $v1: `29`

Example #3:

```
plaintext = "If debugging is the process of removing software bugs, then
programming must be the process of putting them in. -Edsger Dijkstra"

keyphrase = "What is the sum of 12 and 37? The answer, CLEARLY, is 49!"

corpus = "It was the best of times, it was the worst of times, it was the
age of wisdom, it was the age of foolishness, it was the epoch of belief,
it was the epoch of incredulity, it was the season of Light, it was the
season of Darkness, it was the spring of hope, it was the winter of
despair, we had everything before us, we had nothing before us, we were
all going direct to Heaven, we were all going direct the other way - in
short, the period was so far like the present period, that some of its
noisiest authorities insisted on its being received, for good or for evil,
in the superlative degree of comparison only."
```

Resulting `ciphertext`: `"L7 e1iXTTAkT 9K MCu zFxsnGH x7 F2jxZEkT Gy7P0hDo iXTI, VCfk zFqTDhjj4kT jXKP in Mwm zFqsdKG q7 zXNOEkT VCnj Rk. -2eGTuD eAbcKSDt"`

Return value in $v0: `105`

Return value in $v1: `23`

Part 10: Decrypt Ciphertext that was Encrypted using a Homophonic Substitution Cipher

```
int, int decrypt(string plaintext, string ciphertext, string keyphrase,
                 string corpus)
```

This function decrypts a ciphertext message that was encrypted using the homophonic substitution cipher described earlier in the document. The decryption algorithm is very similar to the encryption algorithm. All arguments are assumed to be valid. The buffer for the ciphertext is guaranteed to be large enough to store the null-terminated `plaintext`. Note that the function returns two values. The generated plaintext letters will all be in lowercase.

The function takes the following arguments, in this order:

- `plaintext`: an uninitialized buffer to store the decrypted ciphertext
- `ciphertext`: a null-terminated string to be decrypted using the homophonic substitution cipher
- `keyphrase`: a null-terminated keyphrase string used during the decryption process
- `corpus`: a null-terminated corpus string used during the decryption process

The function implements the following algorithm:

1. Call `to_lowercase` on `corpus`.
2. Allocate at least 26 words' worth of memory on the stack to temporarily store the `counts` array needed in the following step. Be sure to deallocate this memory later.
3. Call `count_lowercase_letters(counts, corpus)`.
4. Allocate at least 27 bytes of memory on the stack to store the `lowercase_letters` string needed in the following step. Be sure to deallocate this memory later.
5. Call `sort_alphabet_by_count(lowercase_letters, counts)`.
6. Allocate at least 63 bytes of memory on the stack to temporarily store the `plaintext_alphabet` string needed in the following step. Be sure to deallocate this memory later.
7. Call `generate_plaintext_alphabet(plaintext_alphabet, lowercase_letters)`.
8. Allocate at least 63 bytes of memory on the stack to temporarily store the `ciphertext_alphabet` string needed in the following step. Be sure to deallocate this memory later.
9. Call `generate_ciphertext_alphabet(ciphertext_alphabet, keyphrase)`.
10. In a loop, decrypt each alphabetical character of `ciphertext` and write the decrypted character into `plaintext`. Each non-alphanumerical character of `ciphertext` should simply be copied to `plaintext`.
11. Null-terminate the `plaintext` string.

Returns in $v0:

- The number of lowercase letters that were written into the `plaintext` buffer during the decryption process.

Returns in $v1:

- The number of non-letters that were written into the `plaintext` buffer during the decryption process.

Additional requirements:

- The function must not write any changes to main memory except as needed.
- The function must call the functions `to_lowercase`, `count_lowercase_letters`, `sort_alphabet_by_count`, `generate_ciphertext_alphabet`, `generate_plaintext_alphabet`, and `index_of`.

Example #1:

```
ciphertext = "Dk5wQ 1Q4RW h yLCO4XnQ 8H4 yaE'3 VpQH6 L4V a 6qGoJ6. -R3n5t
6J9GqIA"
```

```
keyphrase = "I'll have you know that I stubbed my toe last week and only
cried for 20 minutes."

corpus = "When in the Course of human events, it becomes necessary for one
people to dissolve the political bands which have connected them with
another, and to assume among the powers of the earth, the separate and
equal station to which the Laws of Nature and of Nature's God entitle
them, a decent respect to the opinions of mankind requires that they
should declare the causes which impel them to the separation."
```

Resulting `plaintext`: `"never trust a computer you can't throw out a window. -steve wozniak"`

Return value in $v0: `53`

Return value in $v1: `14`

Example #2:

```
ciphertext = "Xjc 1UN4dDn 6xXj jW5vJk WJ ORcK GmKf, PI iO4TVn, mV 0jW3
RwQREy 6mDE xKVlV1 NJ iMGqLk aDPJk aJf 2U8uHk 2Q R40 2jlJkV xK lZ. -3rUT8
RThYijc23"

keyphrase = "What's the difference between ignorance and apathy? I don't
know and I don't care."

corpus = "Call me Ishmael. Some years ago - never mind how long precisely
- having little or no money in my purse, and nothing particular to
interest me on shore, I thought I would sail about a little and see the
watery part of the world. It is a way I have of driving off the spleen and
regulating the circulation."
```

Resulting `plaintext`: `"the trouble with having an open mind, of course, is that people will insist on coming along and trying to put things in it. -terry pratchett"`

Return value in $v0: `111`

Return value in $v1: `29`

Example #3:

```
ciphertext = "L7 e1iXTTAkT 9K MCu zFxsnGH x7 F2jxZEkT Gy7P0hDo iXTI, VCfk
zFqTDhjj4kT jXKP in Mwm zFqsdKG q7 zXNOEkT VCnj Rk. -2eGTuD eAbcKSDt"
```

```
keyphrase = "What is the sum of 12 and 37? The answer, CLEARLY, is 49!"
corpus = "It was the best of times, it was the worst of times, it was the
age of wisdom, it was the age of foolishness, it was the epoch of belief,
it was the epoch of incredulity, it was the season of Light, it was the
season of Darkness, it was the spring of hope, it was the winter of
despair, we had everything before us, we had nothing before us, we were
all going direct to Heaven, we were all going direct the other way - in
short, the period was so far like the present period, that some of its
noisiest authorities insisted on its being received, for good or for evil,
in the superlative degree of comparison only."
```

Resulting `plaintext`: `"if debugging is the process of removing software bugs, then programming must be the process of putting them in. -edsger dijkstra"`

Return value in $v0: `105`

Return value in $v1: `23`

Academic Honesty Policy

Academic honesty is taken very seriously in this course. By submitting your work for grading 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 University's Academic Judiciary.
7. I understand that the penalty for academic dishonesty might not be immediately administered. For instance, cheating on a homework assignment 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 hwk2.asm file for grading:

1. Go to the course website.
2. Click the Submit link for this assignment.
3. Type your SBU ID# on the line provided.
4. Press the button marked **Add file** and follow the directions to attach your file.
5. Hit Submit to submit your file 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.