CSE 220: Systems Fundamentals I

Stony Brook University

Programming Project #2

Spring 2020

Assignment Due: Monday, March 16, 2020 by 11:59 pm

Updates to the Document:

- 3/11/2020: Due data postponed by three days.
- 3/11/2020: Added note to Part IV in red.
- 3/4/2020: Part VI, Example 2. The return value has been updated.

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 strings of arbitrary length.
- Design and implement 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 500,000 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 enqueue 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. 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 `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!

Pac-Man Eating Strings

Your first main task will be to implement a function that solves a funny little problem involving Pac-Man (of arcade game legend) eating ASCII characters instead of dots. As part of this effort you will need to implement a few support functions for working with strings.

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 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 length of 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`
`""`	`0`

Part II: Insert a Character into a String

`int insert(string str, char ch, int index)`

This function takes a null-terminated string (possibly empty), a character, and an integer index, and inserts the character at that index, right-shifting all characters to the right of that index by 1 position. Assuming the insertion index is valid (i.e., in the range 0 through `strlen(str)`, inclusive), the function returns the length of the modified string. If the insertion index is invalid (namely, less than zero or greater than the length of the string), the function returns `-1` and makes no change to the string. The function may assume that it is always provided an extra byte of memory to accommodate the inserted character. The function must also ensure that the final string is properly null-terminated.

The function takes the following arguments, in this order:

- `str`: the starting address of a null-terminated string
- `ch`: a printable ASCII character to be inserted into `str`
- `index`: the index at which to insert `c` in `str`

Returns in `$v0`:

- the length of the updated string on success, or `-1` on error

Additional requirements:

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

Examples:

Function Arguments	**`str`** after Call	**Return Value**
`"abcdefgh\0", 'X', 0`	`"Xabcdefgh\0"`	9
`"abcdefghij\0", 'X', 3`	`"abcXdefghij\0"`	11
`"abcdefghijkl\0", 'X', 12`	`"abcdefghijklX\0"`	13
`"\0", 'X', 0`	`"X\0"`	1
`"abcdefghij\0", 'X', 15`	`"abcdefghij\0"`	-1

Part III: Let's See if Pac-Man Can Eat a String

`(int, int) pacman(string str)` (The function returns two values.)

Pac-Man is a video game character who normally eats dots, power pellets, and the occasional ghost. Recently, he's fallen on hard times and has been reduced to (mainly) eating letters of the alphabet. Pac-Man can eat most letters of the alphabet, but he is unable to digest any of the characters in the word "GHOST"(uppercase or lowercase). When he reaches one of these characters in a string, he loses his appetite and stops eating.

In this part of the assignment you will write a function takes a non-empty, null-terminated string of letters only and determines if Pac-Man can eat the string, or at least part of the string. The function traces Pac-Man's progress through a string of uppercase and lowercase letters (with no spaces, digits, or symbols) and updates the state of the string accordingly. An underscore represents a consumed character, and a less-than sign represent Pac-Man's final position (either at the very beginning or end of the string, or right before the character that stopped him). The function returns the index inside the string where Pac-Man comes to rest and the length of the final string, in that

order.

For example, consider the null-terminated string `"cat\0"`. Pac-Man can eat the `'c'` and the `'a'`, but stops when he reaches `'t'` (because it is one of the letters in "GHOST"). Logically speaking, Pac-Man is sitting in the spot formerly occupied by the `'a'`. Thus, the final string will be `"_<t\0`. The return values will be `1` and `3`, respectively.

As another example, consider the string `"bird\0"`. Pac-Man can eat the entire string. Therefore, the final state of the string will be `"____<\0"`. Note that the string is one character longer than it was before the function was called (there are four underscores) and that the string is properly null-terminated. The function may always assume that an extra byte of memory will provided to accommodate such cases. The return values will be `4` and `5`, respectively, for this example.

One final example: consider the string `"tiger\0"`. Pac-Man can't even eat the first character of the string. The final, updated string will be `"<tiger\0"`. Note that this string is one character longer than the input string. The return values will be `0` and `6`, respectively.

The function takes one argument:

- `str`: a line (string) of characters for Pac-Man to eat

Returns in `$v0`:

- the index of the `'<'` character in the final string

Returns in `$v1`:

- the length of the final string

Additional requirements:

- The function must not write any changes to main memory except as necessary.
- The function must call `insert`.

Examples:

Remember that the string increases in length by `1` in cases where Pac-Man can eat all of the string or none of the string. If Pac-Man can eat only part of the string, then the string's length does not change.

Function Arguments	Updated Argument	Return Values
`"abcdEFIJKLmnpQRUVWXYZ\0"`	`"_____<\0"`	`21, 22`
`"abcdefoABCDEsthnb\0"`	`"_____<oABCDEsthnb\0"`	`5, 17`
`"thisisnotgoingtoendwell\0"`	`"<thisisnotgoingtoendwell\0"`	`0, 24`
`"abracadabraS\0"`	`"_____<S\0"`	`10, 12`

Byte-Pair Encoding and Decoding

In this portion of assignment you will be implementing a simple compression algorithm called byte-pair encoding and its attendant decompression algorithm. Go ahead and ready the very short, linked article to get the idea of

what you'll be doing.

As the name of the algorithm suggests, byte-pair encoding involves replacing (encoding) a pair of adjacent characters from an input string with a different, single character that does not appear in the input. To keeps things manageable in this assignment, we will make some simplifying assumptions about the input and the algorithm:

1. the input consists only of lowercase letters and no other characters
2. pairs of lowercase letters will be replaced by single capital letters
3. due to point #2, at most 26 rounds of substitutions can be made

Encoding is an iterative process that starts at index 0 of the string and finds the adjacent byte-pair that occurs most frequently in the string. As an example, consider the string

```
aabbacbacbacbacbababababcabbacabca\0
```

On the first pass through the string we compute the following frequencies, where frequencies of 0 have been omitted:

```
aa: 1
ab: 7
ac: 5
ba: 9
bb: 2
bc: 2
ca: 3
cb: 4
```

You will note that the input string contains 34 characters and that the sum of the frequencies is 33, the number of byte-pairs in the string. We see that `"ba"` occurs most frequently. Therefore, we will replace all instances of `"ba"` with a single letter, namely, capital letter `'Z'`. As in the Wikipedia article, we will move backwards through the alphabet when choosing substitution letters. After one round of substitutions we now have:

```
aabZcZcZcZcZZZZbcabZcabca\0\0\0\0\0\0\0\0\0\0
```

Each time a single substitution is performed, the characters to the right of substituted character are shifted to the left by one index.

The algorithm performs another round of computing frequencies and inspects *adjacent* pairs of lowercase letters, recomputing the frequencies:

```
aa: 1
ab: 3
bc: 2
ca: 3
```

We have a tie in frequencies for two byte-pairs. In our implementation, if two or more byte-pairs have the same frequency in a given pass through the string, substitute the byte-pair that comes first alphabetically. In this example we will choose to replace `"ab"` with `'Y'` because `"ab"` comes before `"ca"` in alphabetical order:

```
aYZcZcZcZcZZZZbcYZcYca\0\0\0\0\0\0\0\0\0\0\0\0
```

Note that 3 substitutions were performed.

We can continue to perform rounds of substitutions like this until one of two conditions is met:

- there are no pairs of adjacent lowercase letters in the string
- 26 rounds of substitutions have been performed, thereby exhausing all 26 uppercase letters of the alphabet

In this example we can complete two more rounds of substitutions, which yields the final string:

```
aYZcZcZcZcZZZZXYZcYW\0\0\0\0\0\0\0\0\0\0\0\0\0\0
```

The mapping of uppercase letters to lowercase letters is given below:

```
W: ca
X: bc
Y: ab
Z: ba
```

This table of mappings, along with the encoded string, are sufficient to decode (decompress) the compressed string. The algorithm is quite simple: iterate over the encoded string, substituting each uppercase letter with its associated byte-pair. Each time a substitution is performed, the characters of the string to the right of the substitution must be shifted right by one index. Easy peasy!

Part IV: Replace the First Instance of a Byte-Pair with a Single Character

```
int replace_first_pair(string str, char first, char second,
                       char replacement, int start_index)
```

This function takes a non-empty, null-terminated string and searches for the first instance of the characters `first` and `second` (in that order) lying next to each other in the string. The search starts at index `start_index` and moves to the right, towards higher indexes. Upon a successful search, the byte-pair (`first` and `second`) are replaced with the character `replacement`, and the function returns the index of `first`. All characters to the right of the substituted character are shifted to the left by one index. The null-terminator is also shifted, which causes a null-terminator to be created near the end of the string. On an unsuccessful search, the function makes no changes to `str` or anywhere else in memory and returns `-1`.

The function takes the following arguments, in this order:

- `str`: a null-terminated string of ASCII characters
- `first`: a printable ASCII character
- `second`: a printable ASCII character
- `replacement`: a printable ASCII character that will be used to replace a byte-pair consisting of characters `first` and `second`
- `start_index`: the index of `str` at which to start a search for the byte-pair given by `first` and `second`

Returns in `$v0`:

- the first index at which character `first` was found (at or after index `start_index`), or `-1` for an unsuccessful search

Additional requirements:

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

Example #1:

Function Arguments: `"aabbacbacbacbacbababababcab\0", 'b', 'a', '?', 10`

Updated `str` Argument: `"aabbacbacbac?cbababababcab\0\0"`

Return Value: `12`

Example #2:

Function Arguments: `"aabbacbacbacbacbababababcab\0", 'b', 'a', '?', 0`

Updated `str` Argument: `"aab?cbacbacbacbababababcab\0\0"`

Return Value: `3`

Example #3:

Function Arguments: `"aabbacbacbacbacbababababcab\0", 'b', 'a', '?', 3`

Updated `str` Argument: `"aab?cbacbacbacbababababcab\0\0"`

Return Value: `3`

Example #4:

Function Arguments: `"abcdefg\0", 'f', 'g', '?', 0`

Updated `str` Argument: `"abcde?\0\0"`

Return Value: `5`

Example #5:

Function Arguments: `"aabbacbacbacbacbababababcab\0", 'b', 'a', '?', 22`

Updated `str` Argument: `"aabbacbacbacbacbababababcab\0"`

Return Value: `-1`

Example #6:

Function Arguments: `"aabbacbacbacbacbababababcab\0", 'b', 'a', '?', 50`

Updated `str` Argument: `"aabbacbacbacbacbababababcab\0"`

Return Value: `-1`

Example #7:

Function Arguments: `"a\0", 'b', 'a', '?', 0`

Updated `str` Argument: `"a\0"`

Return Value: `-1`

Example #8:

Function Arguments: `"Uppercase Is OK Too\0", 'T', 'o', 'S', 10`

Updated `str` Argument: `"Uppercase Is OK So\0\0"`

Return Value: `16`

Part V: Replace All Instances of a Byte-Pair with the Same Character

`int replace_all_pairs(string str, char first, char second, char replacement)`

This function takes a non-empty, null-terminated string and replaces all instances of the characters `first` and `second` (in that order) lying next to each other in `str`. The search begins at index 0 and proceeds towards the right. Each such byte-pair that is found is replaced immediately with the character `replacement` before continuing to the next character in the string. So, for example, if `str` were `"aaab"`, `first` and `second` were both `'a'`, and `replacement` were `'?'`, the contents of `str` would be changed to `"?ab"`, not `"??b"`.

The function takes the following arguments, in this order:

- `str`: a null-terminated string of ASCII characters
- `first`: a printable ASCII character
- `second`: a printable ASCII character
- `replacement`: a printable ASCII character that will be used to replace byte-pairs consisting of characters `first` and `second`

Returns in `$v0`:

- the number of byte-pair replacements that were performed by the function

Additional requirements:

- The function must not write any changes to main memory except as specified.
- The function must call `replace_first_pair`. The idea is to call `replace_first_pair` repeatedly, in a loop, using the return value to determine where to continue the search (namely, the return value plus 1).

Example #1:

Function Arguments: `"aabbacbacbacbacbababababcab\0", 'b', 'a', 'Z'`

Updated `str` Argument: `"aabZcZcZcZcZZZZbcab\0\0\0\0\0\0\0\0\0"`

Return Value: `8`

Example #2:

Function Arguments: `"bbbabbbabbbabbba\0", 'b', 'b', 'Z'`

Updated `str` Argument: `"ZbaZbaZbaZba\0\0\0\0\0"`

Return Value: `4`

Example #3:

Function Arguments: `"aabbacbacbacbababababcab\0", 'x', 'y', 'Z'`

Updated `str` Argument: `"aabbacbacbacbababababcab\0"`

Return Value: `0`

Example #4:

Function Arguments: `"attattattattatt\0", 'a', 't', 'a'`

Updated `str` Argument: `"atatatatat\0\0\0\0\0\0"`

Return Value: `5`

Example #5:

Function Arguments: `"dinglehopper\0", 'e', 'r', 's'`

Updated `str` Argument: `"dinglehopps\0\0"`

Return Value: `1`

Example #6:

Function Arguments: `"Stony Crony Baloon\0", 'o', 'n', 'a'`

Updated `str` Argument: `"Stay Cray Baloa\0\0\0\0"`

Return Value: `3`

Part VI: Byte-Pair Encode a String

```
int bytepair_encode(string str, unsigned byte[] frequencies,
                    char[] replacements)
```

This function takes a non-empty, null-terminated string and encodes it using the byte-pair encoding scheme described above. The encoded string is stored in `str`, thereby overwriting the original string. The arrays `frequencies` and `replacements` are *uninitialized* buffers (i.e., filled with garbage values) which *must be initialized* by the function by zeroing out the contents. `frequencies` is always 676 (26^2) bytes in size, and `replacements` is always 52 bytes in size. You may assume that no byte-pair appears more than 255 times in the `str` argument. These two arrays are filled with values during the encoding process, but only the `replacements` array will be graded. Specifically, `replacements[2*i]` and `replacements[2*i+1]` contain the byte-pair associated with capital letter #`i`, where A is letter #0 and Z is letter #25. As an example, X is the 24th letter of the alphabet, so the two-letter byte-pair that it is mapped to will be at indices `46` and `47` of `replacements` (indexes are 0-based). The `frequencies` buffer is provided as a convenience to the programmer and can be used as the programmer deems fit, or not used at all.

The function takes the following arguments, in this order:

- `str`: a null-terminated string of ASCII characters
- `frequencies`: an array of 676 bytes that stores frequencies of byte-pairs computed during the encoding process (will not be graded)
- `replacements`: an array of 52 bytes that maps uppercase letters to byte-pairs (will be graded)

Returns in `$v0`:

- the number of byte-pair substitutions performed during encoding (i.e., the sum of all return values from `replace_all_pairs`.

Additional requirements:

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

Example #1:

First Argument: `"stonybrookuniversity\0"`

Updated `str` Argument: `"SoUZTVnWYsXy\0\0\0\0\0\0\0\0\0"`

Updated `replacements` Argument: `"\0\0 \0\0 ... \0\0 st oo ny ku iv it er br"`

(Each of A through R is mapped to two null-terminators, followed by the mappings for letters S through Z.)

Return Value: `8`

Example #2:

First Argument: `"abracadabrahoppenshoppenboppen\0"`

Updated `str` Argument: `"XWcVXWhYpZUYpZbYpZ\0\0\0\0\0\0\0\0\0\0\0\0"`

Updated `replacements` Argument: `"\0\0 \0\0 ... \0\0 sh ad ra ab op en"`

(Each of A through T is mapped to two null-terminators, followed by the mappings for letters U through Z.)

Return Value: `12`

Example #3:

First Argument: `"tooootototttotoototototototo\0"`

Updated `str` Argument: `"ZYoZZXtZZoZZZZZ\0\0\0\0\0\0\0\0\0\0\0\0\0"`

Updated `replacements` Argument: `"\0\0 \0\0 ... \0\0 tt oo to"`

(Each of A through W is mapped to two null-terminators, followed by the mappings for letters X through Z.)

Return Value: `12`

Example #4:

First Argument: `"a\0"`

Updated `str` Argument: `"a\0"`

Updated `replacements` Argument: `"\0\0 \0\0 ... \0\0"`

(The entire array consists of null-terminators.)

Return Value: `0`

Part VII: Replace the First Instance of a Character with a Byte-Pair

```
int replace_first_char(string str, char ch, char first, char second,
                       int start_index)
```

This function takes a non-empty, null-terminated string and searches for the first instance of the character `ch` in `str`, starting at index `start_index` of `str` and moving to the right. Upon a successful search, the instance of `ch` is replace with characters `first` and `second`, which causes all characters to the right of index `start_index` to be shifted to the right by one index. The null-terminator is also shifted. The function may assume that an extra byte of uninitialized memory has been allocated to the right of `str` to accommodate this shift. On an unsuccessful search, the function makes no changes to `str` or anywhere else in memory and returns `-1`.

The function takes the following arguments, in this order:

- `str`: a null-terminated string of ASCII characters
- `ch`: a printable ASCII character
- `first`: a printable ASCII character
- `second`: a printable ASCII character
- `start_index`: the index of `str` at which to start a search for the character `ch`

Returns in `$v0`:

- the first index at which character `ch` was found (at or after index `start_index`), or `-1` for an unsuccessful search

Additional requirements:

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

Example #1:

Function Arguments: `"acadZacabaZcbacZeabZaQ\0", 'Z', 'X', 'Y', 0`

Updated `str` Argument: `"acadXYacabaZcbacZeabZaQ\0"`

Return Value: `4`

Example #2:

Function Arguments: `"acadZacabaZcbacZeabZaQ\0", 'Z', 'X', 'Y', 10`

Updated `str` Argument: `"acadZacabaXYcbacZeabZaQ\0"`

Return Value: `10`

Example #3:

Function Arguments: `"acadZacabaZcbacZeabZaQ\0", 'Z', 'X', 'Y', 35`

Updated `str` Argument: `"acadZacabaZcbacZeabZaQ\0"`

Return Value: `-1`

Example #4:

Function Arguments: `"acadZacabaZcbacZeabZaQ\0", 'a', 'X', 'Y', 0`

Updated `str` Argument: `"XYcadZacabaZcbacZeabZaQ\0"`

Return Value: `0`

Example #5:

Function Arguments: `"acadZacabaZcbacZeabZaQ\0", 'Q', 'X', 'Y', 0`

Updated `str` Argument: `"acadZacabaZcbacZeabZaXY\0"`

Return Value: `21`

Part VIII: Replace All Instances of a Character with the Same Byte-Pair

`int replace_all_chars(string str, char ch, char first, char second):`

This function takes a non-empty, null-terminated string and replaces all instances of the character `ch` found in `str`. The search begins at index 0 and proceeds towards the right. Each such instance of `ch` that is found is replaced immediately with the characters `first` and `second` before continuing to the next character in the string. (Note that the search continues with the character that now lies immediately to the right of the copy of `second` that was inserted into the string.) The replacement and shifting actions are accomplished by a call to `replace_first_char`. The function may assume that enough uninitialized memory has been allocated to the right of `str` to accommodate the new character. However, you should not assume that those bytes have been initialized with null-terminators.

The function takes the following arguments, in this order:

- `str`: a null-terminated string of ASCII characters
- `ch`: a printable ASCII character
- `first`: a printable ASCII character
- `second`: a printable ASCII character

Returns in `$v0`:

- the number of characters replace by the function

Additional requirements:

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

- The function must call `replace_first_char`.

Example #1:

Function Arguments: `"acadZacabaZcbacZeabZaQ\0", 'Z', 'X', 'Y'`

Updated `str` Argument: `"acadXYacabaXYcbacXYeabXYaQ\0"`

Return Value: `4`

Example #2:

Function Arguments: `"QQQQQQ\0", 'Q', 'a', 'b'`

Updated `str` Argument: `"abababababab\0"`

Return Value: `6`

Example #3:

Function Arguments: `"Qabcdefg\0", 'Q', 'X', 'Y'`

Updated `str` Argument: `"XYabcdefg\0"`

Return Value: `1`

Example #4:

Function Arguments: `"abcabcabc\0"`

Updated `str` Argument: `"abbcabbcabbc\0", 'a', 'a', 'b'`

Return Value: `3`

Example #5:

Function Arguments: `"abcabcabc\0", 'x', 'y', 'z'`

Updated `str` Argument: `"abcabcabc\0"`

Return Value: `0`

Part IX: Decode a String Encoded with Byte-Pair Encoding

```
int bytepair_decode(string str, char[] replacements)
```

This function takes a non-empty, null-terminated string and a mapping of byte-pair replacements (using the same scheme as decribed in Part VI), and updates the contents of `str` by applying the mappings provided in `replacements`. The function accomplishes this task by repeatedly calling `replace_all_chars`. The function may assume that a sufficient amount of uninitialized memory has been allocated past the end of `str` to accommodate the substitutions.

The function takes the following arguments, in this order:

- `str`: a null-terminated string of characters that store a byte-encoded message
- `replacements`: a 52-character array that maps byte-pairs to uppercase letters. The array is arranged

according to the scheme described in Part VI.

Returns in `$v0`:

- the number of characters from `str` replaced with character byte-pairs

Additional requirements:

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

Example #1:

`str` Argument: `"SoUZTVnWYsXy\0"`

`replacements` Argument: `"\0\0 \0\0 ... \0\0 st oo ny ku iv it er br"`

Updated `str` Argument: `"stonybrookuniversity\0"`

Return Value: `8`

Example #2:

`str` Argument: `"YpZWXQSeTRVUYpZWXt\0"`

`replacements` Argument: `"\0\0 \0\0 ... \0\0 to nd fr du cy an tm en de ar"`

Updated `str` Argument: `"departmentofredundancydepartment\0"`

Return Value: `14`

Example #3:

`str` Argument: `"ZZZZZZZZa\0"`

`replacements` Argument: `"\0\0 \0\0 ... \0\0 aa"`

Updated `str` Argument: `aaaaaaaaaaaaaaaaa\0""`

Return Value: `8`

Example #4:

`str` Argument: `"ZbZbZbZbZbZY\0"`

`replacements` Argument: `"\0\0 \0\0 ... \0\0 ba ab"`

Updated `str` Argument: `"abbabbabbabbabbabba\0"`

Return Value: `7`

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.