CSE 220: Systems Fundamentals I

Stony Brook University

Homework Assignment #2

Fall 2018

Assignment Due: October 12, 2018 by 11:59 pm

Important Information about CSE 220 Homework Assignments

- Read the entire homework documents twice before starting. Questions posted on Piazza whose answers are clearly stated in the documents will be given lowest priority by the course staff.
- When writing assembly code, try to stay consistent with your formatting and to comment as much as possible. It is much easier for your TAs and the professor to help you if we can quickly figure out what your code does.
- You personally must implement homework assignments in MIPS Assembly language by yourself. You may not write or use a code generator or other tools that write any MIPS code for you. You must manually write all MIPS Assembly code you submit as part of the assignments.
- Do not copy or share code. Your submissions will be checked against other submissions from this semester and from previous semesters.
- You must use the Stony Brook version of MARS posted on Piazza. Do not use the version of MARS posted on the official MARS website. The Stony Brook version has a reduced instruction set, added tools, and additional system calls you will need to complete the homework assignments.
- **For Homework Assignments #2 and later, do not submit a file with the functions/labels `main` or `_start` defined. You are also not permitted to start your label names with two underscores (`__`). You will obtain a zero for an assignment if you do this.**
- Submit your final `.asm` file to Blackboard by the due date and time. Late work will not be accepted or graded. Code that crashes and cannot be graded will earn no credit. No changes to your submission will be permitted once the deadline has passed.

How Your CSE 220 Assignments Will Be Graded

With minor exceptions, all aspects of your homework submissions will be graded entirely through automated means. Grading scripts will execute your code with input values (e.g., command-line arguments, function arguments) and will check for expected results (e.g., print-outs, return values, etc.) For Homework Assignments #2 and later 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 10,000 instructions or fewer. Efficiency is an important aspect of program-

ming. This maximum instruction count will be increased in cases where a complicated algorithm might be necessary, or a large data structure must be traversed.

- Any excess output from your program (debugging notes, etc.) 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.

Assignment Objectives

The primary objectives of this homework assignment are to continue mastering array processing in MIPS, to be able to write functions in MIPS, and to apply MIPS register conventions.

Register Conventions

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

- It is the callee's responsibility to save any `$s` registers it overwrites by saving copies of those registers on the stack and restoring them before returning.
- If a function calls a secondary function, the caller must save `$ra` before calling the callee. In addition, if the caller wants a particular `$a`, `$t` or `$v` register's value to be preserved across the secondary function call, the best practice would be to place a copy of that register in an `$s` register before making the function call.
- A function which allocates stack space by adjusting `$sp` must restore `$sp` to its original value before returning.
- Registers `$fp` and `$gp` are treated as preserved registers for the purposes of this course. If a function modifies one or both, the function must restore them before returning to the caller. There is no reason for your code to touch the `$gp` register, so leave it alone.

The following practices will result in loss of credit:

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

Getting Started

Visit Piazza and download the file `hw2.zip`. Decompress the file and then open `hw2.zip`. Fill in the following information at the top of `hw2.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 `hw2.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 `hw2.asm`.

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 `hw2.asm` file. A submission that contains a `.data` section will receive a score of zero.

Preliminaries

In this assignment, you will familiarize yourself with a feature of the C programming language known as `structs`. A struct is a composite data type that is used to group variables under one name in a contiguous block of memory. You will be reading and writing arrays of structs that represents cars and car repairs. The `*` notation used in the struct definitions below is known as a *pointer* in C. A pointer is simply a memory address. The data type associated with the pointer identifies how the data should be accessed at the memory location that is specified by the pointer. For example, `int *` would signify the address of an integer located somewhere in memory.

The `car` Data Type

The `car` struct stores information about a particular car, identified by its 17-character VIN (vehicle identification number):

```
struct car {
  string vin;     // 4 bytes
  string make;    // 4 bytes
  string model;   // 4 bytes
  short year;     // 2 bytes
  byte features; // 1-byte bit vector
  byte padding;  // 1 byte of zero-padding
};
```

The meaning of each field:

- `vin:` a pointer to the null-terminated string containing the car's 17-character VIN.
- `make:` a pointer to the null-terminated string containing the car's make.

- `model`: a pointer to the null-terminated string containing the car's model.
- `year`: a 16-bit integer (half-word) containing the car's year of manufacture.
- `features`: a bit vector that gives additional information about the car
  - bit 0: is it a convertible (1) or not (0)?
  - bit 1: is it hybrid (1) or not (0)
  - bit 2: are the windows tinted (1) or not (0)?
  - bit 3: is GPS built-in (1) or not (0)?
  - bits 4-7 are unused

Note that the `car` struct contains 16 bytes, including 1 byte of zero-padding.

Here is what a `car` struct might look like in MIPS:

```
# Data declarations from elsewhere in data.asm:
vin_03: .asciiz "1G1AK15F967719757"
make_E: .asciiz "Mersaydeez"
model_D: .asciiz "Road Hog"

# The car struct itself starts at label car_03:
car_03: .word vin_03  # vin_03 is the starting address of "1G1AK15F967719757"
car_03_make_addr: .word make_E   # make_E is the starting address of "Mersaydeez"
car_03_model_addr: .word model_D # model_D is the starting address of "Road Hog"
car_03_year: .byte 175, 7        # the year of manufacture: 175 + 7*256 = 1967
car_03_features: .byte 10        # a bit-vector of features (binary 1010_2)
.byte 0                          # one byte of zero-padding
```

See the included `data.asm` file for sample `car` structs.

While working on the homework you might find it helpful to write a function that prints the contents of a `car` struct to give you a human-readable form of the data structure.

The `repair` Data Type

```
struct repair {
   car* car_ptr;  // 4 bytes
   string desc;   // 4 bytes
   short cost;    // 2 bytes
   short padding; // 2 bytes of zero-padding
}
```

The notation * indicates that the field is a pointer, i.e., the address of a `car` struct.

The meaning of each field:

- `car_ptr`: a pointer to the struct for the car that was repaired.
- `desc`: a pointer to the null-terminated string containing the description of the repair.

- `cost:` a 16-bit unsigned integer containing the cost of the repair.

Note that the `repair` struct contains 12 bytes of data, including 2 bytes of zero-padding.

Here is what a `repair` struct might look like in MIPS:

```
# A data declaration from elsewhere in data.asm:
repair_desc_A: .asciiz "fix cracked windshield"

# The car struct itself starts at label repair_05_car:
repair_05_car: .word car_03               # starting addr. of repaired car struct
repair_05_desc_addr: .word repair_desc_A  # address of repair string
repair_05_cost: .byte 156, 1          # cost to repair the car: 156 + 1*256 = 412
.byte 0, 0                            # two bytes of zero-padding
```

See the included `data.asm` file for sample `repair` structs.

How to Test Your Functions

Numerous `.asm` files have been provided to you for this assignment to help you test your work. However, note the following:

- These tests are not exhaustive. You will need to modify the provided "main" files to test your work more thoroughly.
- These test cases will not be used to grade your work.
- The grading test cases will use different labels than the ones you will find in the `data.asm` file and similar files.
- Do not, under any circumstances, add a `.data` section to your `hs2.asm` file. Doing this will render your submission ungradeable!

Using the provided main files is pretty simple. For example, suppose you wanted to test your `index_of_car` function from Part I of the assignment. First, make sure that in the Settings menu in MARS the option `Initialize Program Counter to global 'main' if defined` is checked. Then, open `index_of_car_main.asm`, assemble it, and then run it. The main file will load the contents of your `hw2.asm` file and call your implementation of the `index_of_car` function.

The main files will provide some basic output, but it is your responsibility to dig deeper into the function outputs to make sure your implementations are correct. These main files will not be used in grading, and you should not submit them for grading. Submit only your `hw2.asm` file to Blackboard.

Finally, make sure that all code required for implementing your functions is included only in the `hw2.asm` file. To make sure that your code is self-contained, try assembling your `hw2.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 `hw2.asm`.

Part I: Search for a Car by Year

```
int index_of_car(car[] cars, int length, int start_index, int year)
```

This function searches an array of `car` structs starting at index `start_index` and returns the index of the *first* car it finds that was manufactured in the given `year`. (That is, when multiple cars of the given year are found, the lowest index $\geq$ `start_index` is returned.) Recall that each `car` struct is 16 bytes in size. Therefore, the starting addresses of consecutive structs are 16 bytes apart. As an example, suppose that `cars[]` starts at address `0x00100000`. `cars[0]` begins at `0x00100000`, `cars[1]` begins at `0x00100010`, `cars[2]` begins at `0x00100020`, etc.

The function takes the following arguments, in this order:

- `cars`: an array of `car` structs
- `length`: the length of the `cars` array (i.e., how many structs are stored in the array)
- `start_index`: the index at which to start the search
- `year`: the year of manufacture

Returns in `$v0`:

- the index of the located car, as described earlier

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

- `length` ≤ 0
- `start_index` < 0
- `start_index` $\geq$ `length`
- `year` < 1885
- No car with the given year was found.

Additional requirements:

- Do not modify the contents of the `cars` array in memory.

Examples:

For the examples below, the `cars` argument is the `all_cars` array provided in the `data.asm` file distributed with the homework. Refer to that file for the particular contents of the array. Be sure to test your function with different arguments (`$a1`, `$a2`, `$a3`) by modifying the `index_of_car_main.asm` file.

Function Call	**Return Value**	**Explanation**
`index_of_car(all_cars, 6, 0, 2017)`	`2`	car found
`index_of_car(all_cars, 6, 2, 2017)`	`2`	car found at/after `start_index`
`index_of_car(all_cars, 6, 3, 2017)`	`-1`	car not found at/after `start_index`
`index_of_car(all_cars, 6, 0, 1950)`	`-1`	no car found for given `year`
`index_of_car(all_cars, 6, -2, 2017)`	`-1`	invalid `start_index`
`index_of_car(all_cars, 0, 1, 2017)`	`-1`	invalid `length`
`index_of_car(all_cars, 6, 0, 1800)`	`-1`	invalid `year`

Part II: Compare Two Strings

`int strcmp(string str1, string str2)`

This function takes two null-terminated strings as arguments and returns an integer that indicates the *lexicographic ordering* of the strings. The function begins by comparing the first character of each string. If they are equal to each other, it continues with the following pairs until the characters differ or until a terminating null-character is reached. Note that the strings can be of different lengths.

The function takes the following arguments, in this order:

- `str1`: a pointer to the first string
- `str2`: a pointer to the second string

Returns in `$v0`:

- the difference between the ASCII values of the first mismatch, if any: `str1[n] - str2[n]`, where `n` is the index of the first mismatch
- `0`: the contents of both strings are identical (including the case where they are both empty strings)
- length of `str1`: `str2` is an empty string but `str1` is non-empty
- negated length of `s2`: `str1` is an empty string but `str2` is non-empty

Additional requirements:

- Do not modify the contents of the argument strings in memory.

Examples:

Function Call	Return Value	Explanation
`strcmp("ABCD", "ABCGG")`	`-3`	First string is smaller; mismatch in middle
`strcmp("WHOOP!", "WHOA")`	`14`	First string is larger; mismatch in middle
`strcmp("Intel", "pentium")`	`-39`	First string is smaller; mismatch at start
`strcmp("STONY", "BROOK")`	`17`	First string is larger; mismatch at start
`strcmp("", "mouse")`	`-5`	First string is empty
`strcmp("lonely guy", "")`	`10`	Second string is empty
`strcmp("Wolfie", "Wolfie")`	`0`	Identical non-empty strings
`strcmp("", "")`	`0`	Two empty strings
`strcmp("happy", "Z")`	`14`	One argument is very short
`strcmp("WOLF", "WOLFIE")`	`-73`	First string is substring of second string
`strcmp("StonyBrook", "Stony")`	`66`	Second string is substring of first string

Part III: Copy a Memory Buffer

`int memcpy(byte *src, byte *dest, int n)`

The `memcpy()` function copies `n` bytes of data from address `src` to address `dest`. We may assume that the `dest` buffer is at least `n` bytes in size.

The function takes the following arguments, in this order:

- `src`: the address to copy bytes from
- `dest`: the address to copy bytes to
- `n`: the number of bytes to copy (must be greater than 0)

Returns:

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

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

- `n` ≤ 0

Additional requirements:

- Do not modify the bytes starting at `src`.
- Only the first `n` bytes memory starting at `dest` may be changed. All other bytes in `dest` must be left unchanged.
- The `src` and `dest` arguments are guaranteed to be valid.

Examples:

Function Call	**Return Value**	**Updated** `dest`
`memcpy("ABCDEFGHIJ", "XXXXXXX", 3)`	`0`	`"ABCXXXX"`
`memcpy("ABCDEFGHIJ", "XXXXXXX", 7)`	`0`	`"ABCDEFG"`
`memcpy("ABCDEFGHIJ", "XXXXXXX", -3)`	`-1`	`"XXXXXXX"`
`memcpy("ABCDEFGHIJ", "XXXXXXX", 0)`	`-1`	`"XXXXXXX"`

Part IV: Insert a `car` Struct into an Array

`int insert_car(car[] cars, int length, car new_car, int index)`

The `insert_car` function inserts a new 16-byte `car` struct at index `index` of array `cars`, shifting the existing car at index `index` and all other cars 16 bytes to the right.

The function takes the following arguments, in this order:

- `cars`: The array of cars to insert into.
- `length`: The number of cars in the array.
- `new_car`: The new car to insert into the array.
- `index`: The index at which the new car should be inserted. If `index` equals `length`, the new car is inserted immediately after the last element of the array.

Returns:

- `$v0`: `0` on success, `-1` if an error occurs.

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

- `length` < 0
- `index` < 0
- `index` $>$ `length`

Additional requirements:

- You may assume that at least 16 bytes of memory has been set aside after the `cars` array to accommodate the new car.
- Do not modify the order of the cars in the array before and after the newly inserted car.
- `insert_car()` must call `memcpy()` to shift the cars in the array.

Examples:

Providing extensive examples in this document is not practical due to the nature of the array being manipulated, so only return values are given in the table below. You are strongly urged to use (and modify!) the `insert_car_main.asm` file provided with the homework materials to test your work. You will note that one complete test case is given in that file. The array after insertion is called `expected_all_cars`. Use your `strcmp` function from earlier in the assignment to compare the contents of `all_cars` (from `data.asm`) with `expected_all_cars` after the `insert_car()` function has been called.

Function Call	Return Value	Explanation
`insert_car(cars_array, 6, new_car, 3)`	`0`	new car inserted at index 3
`insert_car(cars_array, 6, new_car, 0)`	`0`	new car inserted at start of array
`insert_car(cars_array, 6, new_car, 6)`	`0`	new car appended to array
`insert_car(cars_array, -1, new_car, 3)`	`-1`	invalid array length
`insert_car(cars_array, 6, new_car, -1)`	`-1`	invalid insertion index
`insert_car(cars_array, 6, new_car, 8)`	`-1`	insertion index too large

Part V: Find the Most Damaged Car

```
(int, int) most_damaged(car[] cars, repair[] repairs, int cars_length,
                        int repairs_length)
```

The `most_damaged()` function finds the car in `cars` that has the highest total repair cost, returning both the index of that car in `cars` and its total repair cost. If there are multiple cars with the same total repair cost, the function returns the lowest of the indices. A particular VIN can appear in multiple `repair` structs in the `repairs` array. (clarification added 10/7/2018)

The function takes the following arguments, in this order:

- `cars`: an array of `car` structs
- `repairs`: an array of `repair` structs

- `cars_length`: the number of elements in `cars`
- `repairs_length`: the number of elements in `repairs`

Returns:

- `$v0`: the index of the car with the highest total repair cost, or `-1` for an error case
- `$v1`: the total repair cost for the car, or `-1` for an error case

The following are error cases:

- `cars_length` ≤ 0
- `repairs_length` ≤ 0

Additional requirements:

- Do not modify the contents of the `cars` or `repairs` arrays in memory.
- ~~This function must call `strcmp()` to compare VINs.~~ (updated 10/3/2018)

Examples:

For the contents of the `all_cars` array from `data.asm`, the returned index in `$v0` should be `4` and the returned total repair cost in `$v1` should be `587`. You are strongly encouraged to modify the `all_repairs` array to make additional examples yourself. This is very easy to do – simply change the cost of particular `repair` objects. Some examples to consider making: what if the array of repairs has only one item? what if two or more cars have the same total repair cost? what if all the cars have the same repair cost?

Part VI: Sort an Array of `car` Structs

```
int sort(car[] cars, int length)
```

The `sort()` function sorts an array of cars by year. The function must implement odd-even sort (which is a stable sorting algorithm) according to the following pseudocode:

```
def sort(cars):
    sorted = False
    while !sorted:
        sorted = True
        for(i = 1; i < cars.length - 1; i += 2):
            if cars[i] > cars[i + 1]:
                swap cars[i] and cars[i + 1]
                sorted = False

        for(i = 0; i < cars.length - 1; i += 2):
            if cars[i] > cars[i + 1]:
                swap cars[i] and cars[i + 1]
                sorted = False
```

The function must use the stack for temporary space when swapping cars in the array. Again, you must use the stack for temporary space – do not add a `.data` section to your `hw2.asm` file. The following pseudocode can be used to swap cars at indices i and j using the stack:

```
$sp -= 16                     // make space on the stack for 1 car
memcpy(cars[i], $sp, 16)      // copy cars[i] onto the stack
memcpy(cars[j], cars[i], 16) // copy cars[j] to cars[i]
memcpy($sp, cars[j], 16)      // copy the temp car on the stack to cars[j]
$sp += 16
```

The function takes the following arguments, in this order:

- `cars`: an array of cars
- `length`: the number of elements in `cars`

Returns:

- `$v0`: `0` if success, `-1` if an error occurs.

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

- `length` ≤ 0

Additional requirements:

- `sort()` must call `memcpy()` to swap cars in the array.

Example:

The `all_cars` array given in `sort_data.asm` contains the following cars, shown here in a more easily-readable form:

```
JTDKN3DU0D5614628 Fjord Wolfie-Z 2017 1000          <----
2FMDK4JC5DBC37904 Hunday X27 2018 1000
1B4HR28N51F502695 Fjord Escapade 2017 1100          <----
1G1AK15F967719757 Mersaydeez Road Hog 2018 1010
1HGEM1159YL037618 Fjord Elantris 2019 0101
5N1AR2MM2EC648945 Toyoter Raoden 2019 1010
1NKDX90X1WR777109 Honder Sazed 2019 0000
3FAHP0HA5CR371712 Fjord Scadrial 2017 1100          <----
5J6TF2H55AL005521 Fjord Metalborn 2019 1010
1G4GC5G34FF231147 Fjord Terris Roadster 2018 0001
1B7HF16Z0XS322729 Honder Allomancer 2018 0100
1G1PC5SHXC7276485 Toyoter Stormlight 2019 1101
```

After sorting with the even-odd sorting algorithm, the cars will be in this order (given as `sorted_all_cars` in `data.asm`):

```
JTDKN3DU0D5614628 Fjord Wolfie-Z 2017 1000          <----
```

```
1B4HR28N51F502695 Fjord Escapade 2017 1100          <----
3FAHP0HA5CR371712 Fjord Scadrial 2017 1100          <----
2FMDK4JC5DBC37904 Hunday X27 2018 1000
1G1AK15F967719757 Mersaydeez Road Hog 2018 1010
1G4GC5G34FF231147 Fjord Terris Roadster 2018 0001
1B7HF16Z0XS322729 Honder Allomancer 2018 0100
1HGEM1159YL037618 Fjord Elantris 2019 0101
5N1AR2MM2EC648945 Toyoter Raoden 2019 1010
1NKDX90X1WR777109 Honder Sazed 2019 0000
5J6TF2H55AL005521 Fjord Metalborn 2019 1010
1G1PC5SHXC7276485 Toyoter Stormlight 2019 1101
```

Since the even-odd sorting algorithm is *stable*, cars with equal years in the original array appear in the same order in the sorted array. We have put an arrow next to the cars manufactured in 2017 to help illustrate the idea.

Part VII: Find the Most Popular Car Feature

```
int most_popular_feature(car[] cars, int length, nibble features)
```

The `most_popular_feature()` function determines the most popular feature amongst the ones defined in `features`. `features` is a bit vector specifying which of the features to consider when calculating the most popular feature (1 means consider, 0 means do not consider). The bit positions in the `features` argument are assigned these meanings, which correspond with those of the `car` structs:

Bit Position	What a 1 in This Bit Position Means
0	Count the # cars that are convertibles
1	Count the # cars that are hybrids
2	Count the # of car that have tinted windows
3	Count the # of cars that have GPS built-in

The function takes the following arguments, in this order:

- `cars`: an array of `car` structs
- `length`: the number of elements in `cars`
- `features`: the nibble (half-byte) of features that should be considered

Returns:

- `$v0`: a `1` in the bit position corresponding to the most popular of the features we are considering. In the case of a tie, return the higher-valued bit number (i.e. GPS $>$ Tinted windows $>$ Hybrid $>$ Convertible). The examples given below will help to further clarify what this return value means.

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

- `length` ≤ 0
- `features` not in range [1, 15].

- All cars have no features from `features` (i.e. there is no favorite because the count is 0 for all features that are being considered).

Additional requirements:

- Do not modify the contents of the `cars` array in memory.

Examples:

Consider the following function call: `most_popular_feature(all_cars, 6, 7)`. The third argument, 0111_2 in binary, indicates that we will count how many cars are convertibles (bit 0 is 1), how many cars are hybrid (bit 1 is 1), and how many cars have tinted windows (bit 2 is 1). We will not count cars that have a GPS because bit 3 is 0. When we call the function with these arguments for the cars in `data.asm`, the return value is 4, 0100_2 in binary. Bit 2 is associated with tinted windows. This means either that (1) of the three features we considered, tinted windows was the most popular feature, *or*, (2) there was a tie in the counts for tinted windows and some other feature, but we reported tinted windows as the "winner" because it has higher precedence than hybrid engines or a convertible roof.

As another example, suppose we performed a search with 0011_2 as the feature vector, but no cars are hybrids or have convertible roofs. In this case, both of those counts are zeroes, and so the function returns `-1` to indicate an error.

So to summarize, there are only 5 valid return values: -1, 1, 2, 4 or 8. Do you see why?

Function Call	Return Value
`most_popular_feature(all_cars, 6, 15)`	8
`most_popular_feature(all_cars, 6, 7)`	4
`most_popular_feature(all_cars, 6, 3)`	2
`most_popular_feature(all_cars, 6, 5)`	4
`most_popular_feature(all_cars, 6, 1)`	1
`most_popular_feature(all_cars, 6, 0)`	-1
`most_popular_feature(all_cars, -1, 14)`	-1

Part VIII: Compute a VIN's Check Digit

```
char compute_check_digit(string vin, string map, string weights,
                         string transliterate_str)
```

The `compute_check_digit()` function computes the check digit for a valid VIN according to the following pseudocode:

```
def transliterate(ch, transliterate_str):
    return transliterate_str.index_of(ch) % 10

def compute_check_digit(vin, map, weights, transliterate_str):
    sum = 0
    for (i = 0; i < 17; i++):
        sum += transliterate(vin.charAt(i), transliterate_str) *
```

```
        map.index_of(weights.char_at(i))

    return map.char_at(sum % 11)
```

Implementing separate functions for `transliterate`, `char_at`, and `index_of` is not required, but is recommended to make your code easier to debug.

The function takes the following arguments, in this order:

- `vin`: a VIN
- `map`: the string `"0123456789X"`, whose address is passed to the function via `$a1`
- `weights`: the string `"8765432X098765432"`, whose address is passed to the function via `$a2`
- `transliterate_str`: the string `"0123456789.ABCDEFGH..JKLMN.P.R..STUVWXYZ",`, whose address is passed to the function via `$a3`

Returns:

- `$v0`: The check digit of the VIN given as an ASCII character. You do not need to do any error-checking for this function.

Additional requirements:

- Do not modify `vin` in memory.

Examples:

In the table below, `transliterate_str` has been abbreviated `trs` to save space.

Function Call	**Return Value**
`compute_check_digit("JTDKN3DU0D5614628", map, weights, trs)`	0
`compute_check_digit("1B4HR28N01F502695", map, weights, trs)`	5
`compute_check_digit("1HGEM1150YL037618", map, weights, trs)`	9
`compute_check_digit("1FTDF15N0KNB73611", map, weights, trs)`	3
`compute_check_digit("1M2P198C0JW002996", map, weights, trs)`	5

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 im-

portance 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 `hw2.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 on the link for this assignment.
3. Click the "Browse My Computer" button and locate the `hw2.asm` file. Submit only that one `.asm` file.
4. Click the "Submit" button to submit your work for grading.

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

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