CSE 220: Systems Fundamentals I

Stony Brook University

Programming Project #5

Spring 2020

Assignment Due: Sunday, May 10th, 2020 by 11:59 pm

Updates to the Document:

- none yet

Learning Outcomes

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

- Implement non-trivial algorithms that require conditional execution and iteration.
- Design and code functions that implement the MIPS assembly register conventions.
- Implement algorithms that process linked lists of structs.

Getting Started

Visit Piazza and download the file `proj5.zip`. Decompress the file and then open `proj5.zip`. Fill in the following information at the top of `proj5.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 `proj5.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 `proj5.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 `proj5.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 1,000,000 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 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 enqueue the contents of your `proj5.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 `proj5.asm` for grading. Make sure that all code required for implementing your functions is included in the `proj5.asm` file. To make sure that your code is self-contained, try assembling your `proj5.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 `proj5.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!

Linked Lists

For this assignment you, will be implementing linked lists of integers that support *variants* of some operations of Java's `ArrayList` class, but restricted to collections of `int` elements. As with the Java `ArrayList`, element indices in our data structure start at 0. You will be working with two data types:

```
struct IntArrayList {
    int size;  # 4-byte unsigned integer; how many items are in the list
    IntNode* head;  # 4-byte address of first node in list
}

struct IntNode {
    int num;   # 4-byte signed word; the data value stored in the node
    IntNode* next;  # 4-byte address of next node in list
}
```

In an `IntArrayList`, the `head` field stores the address of the first `IntNode` in the linked list. Remember that a memory address is always an unsigned integer. Similarly, in an `IntNode`, the `next` field stores the address of the next `IntNode` in the linked list. When an `IntArrayList` is empty, `head` must have the value `0`, which represents a null pointer. Similarly, the `next` field of the last `IntNode` in a linked list must have the value `0`.

Dynamic Memory Allocation

To store data in the system heap, MARS provides system call #9, which is called sbrk. For example, to allocate `N` bytes of memory, where `N` is a positive integer literal, we would write this code:

```
li $a0, N
li $v0, 9
syscall
```

To allocate enough memory for a new `IntNode`, we would give `8` in place of `N`.

When the system call completes, the address of the newly-allocated memory buffer will be available in `$v0`. The address will be on a word-aligned boundary, regardless of the value passed through `$a0`. Unfortunately, there is no way in MARS to de-allocate memory to avoid creating memory leaks. The run-time systems of Java, Python and some other languages take care of freeing unneeded memory blocks with garbage collection, but assembly languages, C/C++, and other languages put the burden on the programmer to manage memory. You will learn more about this in CSE 320.

The *Hearts* Card Game

In the later parts of the assignment, you will use the above data types to implement an almost-playable version of the Hearts card game. We will encode a playing card as a 4-byte string, which conveniently fits in exactly one word of memory. This means we can store a card using the `num` field of an `IntNode` struct.

Bytes 0–3 of a playing card word will contain:

Byte #0: `'U'` or `'D'`, to encode whether a card is face-up or face-down, respectively

Byte #1: one of `'2'`, `'3'`, ..., `'9'`, `'T'`, `'J'`, `'Q'`, `'K'` or `'A'`, to represent the *ranks* 2 through 10, Jack, Queen, King and Ace, respectively. Aces are high cards in this game.

Byte #2: one of `'C'`, `'D'`, `'H'` or `'S'`, to represent the *suits* Clubs, Diamonds, Hearts and Spades, respectively

Byte #3: `0`, to serve as a null-terminator

For example, the null-terminated `"U2C\0"` represents the face-up, 2 of Clubs. It would be encoded as an integer as `0x00433255` because:

- $55_{16} = 85_{10}$, which is the ASCII code for `'U'`;
- $32_{16} = 50_{10}$, which is the ASCII code for `'2'`;
- $43_{16} = 62_{10}$, which is the ASCII code for `'C'`; and
- 00_{16}, which is the ASCII code for the null-terminator.

Part 1: Initialize a New `IntArrayList`

```
void init_list(IntArrayList* list)
```

This function accepts a pointer to a pre-allocated, 8-byte, word-aligned block of uninitialized memory and, treating the memory block as an uninitialized `IntArrayList` struct, sets the `size` and `head` fields to `0`.

The function takes one argument:

- `list`: the starting address of an 8-byte, word-aligned, uninitialized block of memory

Additional requirements:

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

Part 2: Append an Integer to an `IntArrayList`

```
int append(IntArrayList* list, int num)
```

This function appends the integer `num` to the end of the `IntArrayList` called `list`. The function is responsible for allocating memory on the heap for the new `IntNode` object and initializing the new node accordingly. The new node's `next` is set to the null-pointer. The size of the list is incremented by 1.

The function takes the following arguments, in this order:

- `list`: the starting address of a valid `IntArrayList`. The list might be empty.
- `num`: an integer

Returns in `$v0`:

- the size of the list after appending `num`

Additional requirements:

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

A Note on Examples

The sample linked lists provided throughout the document are keyed to "args" files that are provided with the assignment. For example, the file `append_args.asm` contains the function arguments for the examples in this section. Each node is assigned a random ID number to help distinguish the nodes from each other. *These random ID numbers have no meaning whatsoever. You will not need to assign ID numbers to nodes.*

Example #1: Append an item to an empty list.

```
list = empty
num = 94
```

Return value: `1`

Updated list, after function call:

```
size: 1
contents: 94 (node653)
```

Example #2: Append an item to a short list.

```
list = 962 (node761) -> 762 (node112) -> 978 (node814) -> 526 (node248) ->
       402 (node168)
num = 37
```

Return value: `6`

Updated list, after function call:

```
size: 6
contents: 962 (node761) -> 762 (node112) -> 978 (node814) -> 526 (node248) ->
          402 (node168) -> 37 (node130)
```

Part 3: Insert an Integer into an `IntArrayList`

```
int insert(IntArrayList* list, int num, int index)
```

This function inserts the integer `num` at index `index` of the `IntArrayList` list. The function is responsible for allocating memory on the heap for the new `IntNode` object and initializing the new node accordingly. The links of the list are updated as necessary to link-in the new node. The size of the list is incremented by 1. If `index` equals the size of the list, then the new integer is simply appended to the list. If `index` is negative or greater than the size of the list, no changes are made to the list data structure, and the function returns `-1`.

The function takes the following arguments, in this order:

- `list`: the starting address of a valid `IntArrayList`. The list might be empty.
- `num`: an integer
- `index`: the index at which to insert `num`

Returns in `$v0`:

- the size of the list after appending `num`; or
- `-1` if `index` < 0 or `index` is greater than the size of the list

Additional requirements:

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

Example #1: Add an item to an empty list.

```
list = empty
num = 94
index = 0
```

Return value: `1`

Updated list, after function call:

```
size: 1
contents: 94 (node653)
```

Example #2: Add an item to the middle of a non-empty list.

```
list = 962 (node761) -> 762 (node112) -> 978 (node814) -> 526 (node248) ->
       402 (node168) -> 293 (node130)
num = 24
index = 3
```

Return value: `7`

Updated list, after function call:

```
size: 7
contents: 962 (node761) -> 762 (node112) -> 978 (node814) -> 24 (node428) ->
          526 (node248) -> 402 (node168) -> 293 (node130)
```

Example #3: Add an item to the end of a non-empty list.

```
list = 59 (node69) -> 332 (node209) -> 655 (node147) -> 863 (node140) ->
       140 (node940) -> 846 (node21)
num = 43
index = 6
```

Return value: `7`

Updated list, after function call:

```
size: 7
contents: 59 (node69) -> 332 (node209) -> 655 (node147) -> 863 (node140) ->
          140 (node940) -> 846 (node21) -> 43 (node332)
```

Part 4: Retrieve the Integer in an `IntArrayList` at an Index

```
(int, int) get_value(IntArrayList* list, int index)
```

This function retrieves the value at the given index in the `IntArrayList` referenced by `list`. If the given list is empty, or `index` is negative, or `index` is greater than or equal to the list's size, the function returns `-1` in both `$v0` and `$v1`. Otherwise, the function returns `0` in `$v0` and the integer stored at index `index` in `$v1`.

The function takes the following arguments, in this order:

- `list`: the starting address of a valid `IntArrayList`. The list might be empty.
- `index`: the index of the list from which to read the value

Returns in `$v0`:

- `0` if `index` is a valid index for `list`, or
- `-1` for any of the error conditions explained above

Returns in `$v1`:

- the integer value stored at index `index` in the provided list, or
- `-1` for any of the error conditions explained above

Additional requirements:

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

Example #1: Attempt to get an item from an empty list.

```
list = empty
index = 0
```

Return values: `(-1, -1)`

Example #2: Get an item at index 0.

```
list = 386 (node566) -> 407 (node985) -> 568 (node381) -> 496 (node275) ->
       9 (node393)
index = 0
```

Return values: `(0, 386)`

Example #3: Get an item at an index in the middle of a list.

```
list = 303 (node534) -> 814 (node675) -> 717 (node986) -> 790 (node726) ->
       595 (node294) -> 842 (node256) -> 779 (node361)
index = 3
```

Return values: `(0, 790)`

Part 5: Change the Integer Stored in an `IntArrayList` at an Index

```
(int, int) set_value(IntArrayList* list, int index, int num)
```

This function sets the value at the given index in the `IntArrayList` referenced by `list` to the new value `num`. If the given list is empty, or `index` is negative, or `index` is greater than or equal to the list's size, the function returns `-1` in both `$v0` and `$v1`. Otherwise, the function returns `0` in `$v0` and the old value at index `index` in `$v1`.

The function takes the following arguments, in this order:

- `list`: the starting address of a valid `IntArrayList`. The list might be empty.
- `index`: the index of the list at which to set the value
- `num`: the new integer value to write at the given index

Returns in `$v0`:

- `0` if `index` is a valid index for `list`, or
- `-1` for any of the error conditions explained above

Returns in `$v1`:

- the current integer value stored at index `index` in the provided list, before changing it to `num`; or
- `-1` for any of the error conditions explained above

Additional requirements:

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

Example #1: Attempt to set an item in an empty list.

```
list = empty
```

```
index = 0
num = 4
```

Return values: `(-1, -1)`

Updated list, after function call:

```
size: 0
contents: empty
```

Example #2: Set an item at index 0.

```
list = 627 (node929) -> 820 (node371) -> 714 (node86) -> 72 (node18) ->
       3 (node705)
index = 0
num = 15
```

Return values: `(0, 627)`

Updated list, after function call:

```
size: 5
contents: 15 (node929) -> 820 (node371) -> 714 (node86) -> 72 (node18) ->
          3 (node705)
```

Example #3: Set an item at an index in the middle of a list.

```
list = 622 (node414) -> 684 (node599) -> 120 (node66) -> 520 (node153) ->
       56 (node136) -> 496 (node143) -> 624 (node116)
index = 2
num = 26
```

Return values: `(0, 120)`

Updated list, after function call:

```
size: 7
contents: 622 (node414) -> 684 (node599) -> 26 (node66) -> 520 (node153) ->
          56 (node136) -> 496 (node143) -> 624 (node116)
```

Part 6: Find the Index of an Integer Stored in an `IntArrayList`

```
int index_of(IntArrayList* list, int num)
```

This function returns the index of the leftmost item of `list` equal to `num`. If the list is empty or `num` is not present in the list, the function returns `-1`.

The function takes the following arguments, in this order:

- `list`: the starting address of a valid `IntArrayList`. The list might be empty.
- `num`: the number to search for in `list`

Returns in `$v0`:

- the index of `num` in the list, or
- `-1` if the list is empty or `num` is not in the list

Additional requirements:

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

Example #1: Attempt to get the index of an item from an empty list.

```
list = empty
num = 12
```

Return value: `-1`

Example #2: Get the index of the item at the beginning of the list.

```
list = 386 (node566) -> 407 (node985) -> 568 (node381) -> 496 (node275) ->
       9 (node393)
num = 386
```

Return value: `0`

Example #3: Get the index of an item in the middle of a list.

```
list = 303 (node534) -> 814 (node675) -> 717 (node986) -> 790 (node726) ->
       595 (node294) -> 842 (node256) -> 779 (node361)
num = 842
```

Return value: `5`

Example #4: Get the index of leftmost instance of `num`.

```
list = 34 (node970) -> 887 (node622) -> 232 (node123) -> 493 (node285) ->
       232 (node762) -> 887 (node347) -> 34 (node273) -> 232 (node300)
num = 232
```

Return value: `2`

Part 7: Remove an Integer Stored in an `IntArrayList`

```
(int, int) remove(IntArrayList* list, int num)
```

This function removes the leftmost occurrence of `num` in the given list. If the list is empty or `num` is not present in the list, the function returns `-1` in both `$v0` and `$v1`. Otherwise, the function returns `0` in `$v0` and the index of the leftmost occurrence of `num` in `list`.

The function takes the following arguments, in this order:

- `list`: the starting address of a valid `IntArrayList`. The list might be empty.
- `num`: the number to search for in `list`

Returns in `$v0`:

- `0` if `num` is in the list, or
- `-1` for any of the error conditions explained above

Returns in `$v1`:

- the index of the leftmost occurrence of `num` in `list`, or
- `-1` for any of the error conditions explained above

Additional requirements:

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

Example #1: Attempt to remove an item from an empty list.

```
list = empty
num = 12
```

Return values: `(-1, -1)`

Updated list, after function call:

```
size: 0
contents: empty
```

Example #2: Remove the item at index 0.

```
list = 386 (node566) -> 407 (node985) -> 568 (node381) -> 496 (node275) ->
       9 (node393)
num = 386
```

Return values: `(0, 0)`

Updated list, after function call:

```
size: 4
contents: 407 (node985) -> 568 (node381) -> 496 (node275) -> 9 (node393)
```

Example #3: Remove an item from the middle of a list.

```
list = 303 (node534) -> 814 (node675) -> 717 (node986) -> 790 (node726) ->
       595 (node294) -> 842 (node256) -> 779 (node361)
```

```
num = 790
```

Return values: `(0, 3)`

Updated list, after function call:

```
size: 6
contents: 303 (node534) -> 814 (node675) -> 717 (node986) -> 595 (node294) ->
          842 (node256) -> 779 (node361)
```

Example #4: Remove the index of leftmost instance of `num`.

```
list = 34 (node970) -> 887 (node622) -> 232 (node123) -> 493 (node285) ->
       232 (node762) -> 887 (node347) -> 34 (node273) -> 232 (node300)
num = 232
```

Return values: `(0, 2)`

Updated list, after function call:

```
size: 7
contents: 34 (node970) -> 887 (node622) -> 493 (node285) -> 232 (node762) ->
          887 (node347) -> 34 (node273) -> 232 (node300)
```

Part 8: Initialize a Deck of Playing Cards

```
IntArrayList* create_deck()
```

Through a series of calls to `append`, this function allocates the memory needed for a 52-node `IntArrayList` to represent a sorted deck of face-down cards, appends the cards to the list as needed, and returns a pointer to the constructed `IntArrayList`. The list's `size` field is set to 52. The `num` fields must be initialized with values to represent cards in the following order, starting at index 0 and ending at index 51:

- 2 of Clubs, 2 of Diamonds, 2 of Hearts, 2 of Spades
- 3 of Clubs, 3 of Diamonds, 3 of Hearts, 3 of Spades
- etc.
- King of Clubs, King of Diamonds, King of Hearts, King of Spades
- Ace of Clubs, Ace of Diamonds, Ace of Hearts, Ace of Spades

Below are three simple visualizations of the contents of the list. *The values must appear in the list in this order.*

```
as decimal ints:    4403780 4469316 4731460 5452356
                    4404036 4469572 4731716 5452612 ...
                    4410180 4475716 4737860 5458756
                    4407620 4473156 4735300 5456196
as hex ints:        00433244 00443244 00483244 00533244
                    00433344 00443344 00483344 00533344 ...
```

```
                      00434b44 00444b44 00484b44 00534b44
                      00434144 00444144 00484144 00534144
printed as strings: D2C D2D D2H D2S
                      D3C D3D D3H D3S ...
                      DKC DKD DKH DKS
                      DAC DAD DAH DAS
```

The easiest way to test this function is to print out and inspect the contents of the `IntArrayList` returned by the function.

Returns in `$v0`:

- a pointer to the `IntArrayList` that was initialized as described above

Additional requirements:

- The function must not write any changes to main memory, except as necessary.
- The function must call `init_list` and `append`.

Part 9: Draw a Card from a List of Playing Cards

`(int, int) draw_card(IntArrayList* cards)`

This function simulates the drawing of the *top card* from a list of cards. The cards might be face-up, face-down or a mixture of both. The *top card* is the first card in the list, i.e., at index 0. Assuming the list of cards is non-empty, the function removes the card from the list and returns `0` in `$v0` and the 4-byte number that encodes the card in `$v1`. The size of the list is decremented by 1. If the `cards` list is empty, the function returns `-1` in both `$v0` and `$v1`.

The function takes one following argument:

- `cards`: the address of an `IntArrayList` that represents a collection of playing cards encoded as described on page 5. Every integer in the list's `IntNode` structs is guaranteed to represent a valid playing card. The list might be empty.

Returns in `$v0`:

- `0` if a card was successfully removed from the list, or
- `-1` if the input list is empty

Returns in `$v1`:

- the integer that encodes the card (i.e., the `num` field of the list's head node)
- `-1` if the input list is empty

Additional requirements:

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

Example #1: Attempt to draw a card from an empty list.

```
list = empty
```

Return values: `(-1, -1)`

Example #2: Draw a card from a non-empty list.

```
list = D9H D4S DQH D2S DTH
```

Return values: `(0, 4733252)` (encodes the card `"D9H"`)

Part 10: Deal Cards to a Group of Players

```
int deal_cards(IntArrayList* deck, IntArrayList** players, int num_players,
               int cards_per_player)
```

This function simulates the dealing of `cards_per_player` cards to each player in a group of `num_players` players. The deck of cards can be of arbitrary length and may even contain duplicates. (However, every card will be stored in its own, unique `IntNode` struct.) Initially, the cards in the deck are assumed to be face-down. As each card is dealt to a player, the card is turned face-up. In other words, the card's `num` field must be adjusted to ensure that it now represents a face-up card. Cards are drawn from the deck one-by-one (via `draw_card`) and appended to the players' card lists one at a time (via `append`). The first card is dealt to `players[0]`, the second to `players[1]`, the third to `players[2]`, etc., wrapping around back to `players[0]`. Cards are dealt until each player receives `cards_per_player` cards or the entire deck is exhausted, whichever comes first. It is possible that some players will be dealt more cards than others.

The notation `**` denotes a *double-pointer*, a pointer to a pointer. In this program, the double-pointer signifies an array of pointers. For example, suppose `num_players` is `3`. Then, `players` is fundamentally an array of pointers to `IntArrayList` structs:

```
    players[0] == address of IntArrayList #0 struct

    players[1] == address of IntArrayList #1 struct

    players[2] == address of IntArrayList #2 struct
```

Each of these `IntArrayLists` is then a regular list of integers. Think carefully about the series of `lw` instructions that are needed to access the contents of each `IntArrayList`.

The function takes the following arguments, in this order:

- `deck`: the address of an `IntArrayList` that represents a collection of playing cards encoded as described on page 5. Every integer in the list's `IntNode` structs is guaranteed to represent a valid playing card. The list might be empty.
- `players`: an array of pointers to *initialized* `IntArrayList` structs
- `num_players`: the number of `IntArrayPoints` pointers in `players`

- `cards_per_player`: the number of cards to deal to each player

Returns in `$v0`:

- the total number of cards dealt to the player, or
- `-1` if the deck is empty, or `num_players` is less than or equal to 0, or `cards_per_player` is less than 1

Additional requirements:

- The function must not write any changes to main memory, except as necessary.
- The function must call `draw_card` and `append`.

Example #1: Deal three cards to four players.

```
deck = D3S D3C D7C DJD D2C D4S DAC DAS D7H D6S DKC
       D4C DKD D8H DQS D2S D7S DTD D5H D9D
players =
    Player #0: 0 0
    Player #1: 0 0
    Player #2: 0 0
    Player #3: 0 0
num_players = 4
cards_per_player = 3
```

Return value: `12`

Updated `deck: DKD D8H DQS D2S D7S DTD D5H D9D`

Updated `players`:

```
    Player #0: U3S U2C U7H
    Player #1: U3C U4S U6S
    Player #2: U7C UAC UKC
    Player #3: UJD UAS U4C
```

Example #2: Deal five cards to six players.

```
deck = DAS DKC DAH D5C D6S DJH D7C D2D DAC D4D D3C DAD DQC DTH
       D2H D5S DTC DQD D4S DKD D3S D9S D6D DJD D6H D2C D3H
players =
    Player #0: 0 0
    Player #1: 0 0
    Player #2: 0 0
    Player #3: 0 0
    Player #4: 0 0
    Player #5: 0 0
num_players = 6
```

`cards_per_player = 5`

Return value: `27`

Updated `deck: empty`

Updated `players`:

```
    Player #0: UAS U7C UQC U4S U6H
    Player #1: UKC U2D UTH UKD U2C
    Player #2: UAH UAC U2H U3S U3H
    Player #3: U5C U4D U5S U9S
    Player #4: U6S U3C UTC U6D
    Player #5: UJH UAD UQD UJD
```

Part 11: Return the Point Value of a Card in the *Hearts* Card Game

`int card_points(int card)`

This function returns the number of points that the given card is worth in *Hearts*. The scoring rule is very simple: any Hearts card (regardless of rank) is worth 1 point; the Queen of Spades is worth 13 points.

The function takes the following arguments, in this order:

- `card`: an integer that represents a playing card under the encoding scheme described on page 5

Returns in `$v0`:

- the number of points the card is worth, assuming `num` encodes a valid vard; or
- `-1` if `num` does not encode a valid playing card

Additional requirements:

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

Examples:

Function Argument	Return Value	Explanation
`5458500`	`0`	Face-down Jack of Spades
`4731733`	`1`	Face-up Three of Hearts
`5460309`	`13`	Face-up Queen of Spades
`1234567`	`-1`	Invalid input

Part 12: Simulate a Partial or Complete Game of *Hearts*

`int simulate_game(IntArrayList* deck, IntArrayList** players, int num_rounds)`

This function simulates the first `num_rounds` rounds of a game of *Hearts*. The function may assume that `num_rounds` is in the range $1 - 13$. The deck of cards provided as the first argument is a shuffled, 52-card deck of standard, face-down playing cards with no duplicates. The `players` array always points to exactly four uninitialized `IntArrayList` objects, one per player. We will imagine that the players are numbered 0 through 3.

The function begins by initializing the four player `IntArrayLists` via `init_list`. Then, the entire deck of cards is dealt to the players via `deal_cards`. As in the standard rules, the first player to take a turn is the one holding the 2 of Clubs, which he discards (removes) from his hand. Now, in round-robin order, each remaining player takes a turn by playing the first Club card in his hand. (This is a deviation from the normal rules to eliminate choices in the simulation.) As required by the standard rules of the game, the player who played the card of highest rank earns the points for that round. It is possible that a player has no Clubs in his hand. In that case, he plays the first card in his hand.

As an example, suppose Player #1 holds the 2 of Clubs. Therefore, Player #1 will lead off the game, followed by Players 2, 3 and 0, in that order:

```
Player 1's hand: 4H JD 8S AH 2C 9D 4S 5S 7D KD 3H 6S 3C   Played 2C
Player 2's hand: 7H AD 9C 2H 5H 5D KH 9S TS 6H 6D 4D AS   Played 9C
Player 3's hand: 7C QD 3D JH 8H 2S 8D QS TH 3S JC TD QH   Played 7C
Player 0's hand: 8C 2D TC KC 9H 7S JS 6C 4C 5C KS QC AC   Played 8C
```

Player #2 played the highest-ranked Clubs card, so he will play the first card in the next round. Except for the first round, the player who plays the first card of the round will simply play the first card in his hand. This is accomplished by calling `draw_card` on the player's hand. The other three players must play the leftmost card in their hands of that same suit. (Again, this is a deviation from the normal rules to eliminate choices in the simulation.) If a player cannot play a card of the round's suit, then we call `deal_card` to play the first card in his hand. Play continues in round-by-round fashion until `num_rounds` of play have completed. Note that, once the first player has taken his turn, we iterate over the remaining players, wrapping around to index 0 of `players` when necessary.

The function takes the following arguments, in this order:

- `deck`: the address of an `IntArrayList` that represents a collection of 52, face-down playing cards encoded as described on page 5
- `players`: an array of pointers to *uninitialized* `IntArrayList` structs
- `num_rounds`: the number of rounds of play

Returns in `$v0`:

- an integer that encodes the final scores of all four players. Specifically, byte i encode player #i's score. Remember that our MIPS simulator is little-endian.

Additional requirements:

- The function must not write any changes to main memory, except as necessary.
- The function must call `init_list`, `deal_cards`, `index_of`, `remove`, `draw_card`, `card_points`. It's up to you to figure out when and where to call these functions in a sensible manner.

Example #1: Simulate game of four players for 5 rounds.

```
deck = D8C D4H D7H D7C D2D DJD DAD DQD DTC D8S D9C D3D DKC DAH
       D2H DJH D9H D2C D5H D8H D7S D9D D5D D2S DJS D4S DKH D8D
       D6C D5S D9S DQS D4C D7D DTS DTH D5C DKD D6H D3S DKS D3H
       D6D DJC DQC D6S D4D DTD DAC D3C DAS DQH
players =
    Player #0: uninitialized
    Player #1: uninitialized
    Player #2: uninitialized
    Player #3: uninitialized
num_rounds = 5
```

Return value: `0x04000300`

Player final scores: `0 3 0 4`

Updated `players`:

```
    Player #0: UKC UJS U6C U4C U5C UKS UQC UAC
    Player #1: U9D U4S U5S U7D UKD U3H U6S U3C
    Player #2: U5H U5D UKH UTS U6H U6D U4D UAS
    Player #3: U3D U8D UQS UTH U3S UJC UTD UQH
```

Full round-by-round play:

```
Player #1 holds the 2C and will lead off the game.

Round #1:
   Player 1's hand: 4H JD 8S AH 2C 9D 4S 5S 7D KD 3H 6S 3C   Played 2C
   Player 2's hand: 7H AD 9C 2H 5H 5D KH 9S TS 6H 6D 4D AS   Played 9C
   Player 3's hand: 7C QD 3D JH 8H 2S 8D QS TH 3S JC TD QH   Played 7C
   Player 0's hand: 8C 2D TC KC 9H 7S JS 6C 4C 5C KS QC AC   Played 8C
Round score: 0 point(s)  Points assigned to player #2.

Round #2:
   Player 2's hand: 7H AD 2H 5H 5D KH 9S TS 6H 6D 4D AS   Played 7H
   Player 3's hand: QD 3D JH 8H 2S 8D QS TH 3S JC TD QH   Played JH
   Player 0's hand: 2D TC KC 9H 7S JS 6C 4C 5C KS QC AC   Played 9H
   Player 1's hand: 4H JD 8S AH 9D 4S 5S 7D KD 3H 6S 3C   Played 4H
Round score: 4 point(s)  Points assigned to player #3.

Round #3:
   Player 3's hand: QD 3D 8H 2S 8D QS TH 3S JC TD QH   Played QD
   Player 0's hand: 2D TC KC 7S JS 6C 4C 5C KS QC AC   Played 2D
   Player 1's hand: JD 8S AH 9D 4S 5S 7D KD 3H 6S 3C   Played JD
   Player 2's hand: AD 2H 5H 5D KH 9S TS 6H 6D 4D AS   Played AD
Round score: 0 point(s)  Points assigned to player #2.
Round #4:
   Player 2’s hand: 2H 5H 5D KH 9S TS 6H 6D 4D AS   Played 2H
   Player 3’s hand: 3D 8H 2S 8D QS TH 3S JC TD QH   Played 8H
   Player 0’s hand: TC KC 7S JS 6C 4C 5C KS QC AC   Played TC
   Player 1’s hand: 8S AH 9D 4S 5S 7D KD 3H 6S 3C   Played AH
Round score: 3 point(s)  Points assigned to player #1.

Round #5:
   Player 1’s hand: 8S 9D 4S 5S 7D KD 3H 6S 3C   Played 8S
   Player 2’s hand: 5H 5D KH 9S TS 6H 6D 4D AS   Played 9S
   Player 3’s hand: 3D 2S 8D QS TH 3S JC TD QH   Played 2S
   Player 0’s hand: KC 7S JS 6C 4C 5C KS QC AC   Played 7S
Round score: 0 point(s)  Points assigned to player #2.
```

Example #2: Simulate game of four players for 8 rounds.

```
deck = DTC D6D D3H D3D D7H DAS D5D D2S DQS D4S DKC D9C D2D D7S
       D6S DJC D5H D6H D4H D2C DQC D8S D3S DAC D3C D7C DTD DKH
       D5S D8C D7D DTS D2H D8D D6C D5C D4D D9S DQD D8H DJD DQH
       D9H DKD DAD DAH DKS DTH D9D D4C DJS DJH
players =
    Player #0: uninitialized
    Player #1: uninitialized
    Player #2: uninitialized
    Player #3: uninitialized
num_rounds = 8
```

Return value: `0x04001100`

Player final scores: `0 17 0 4`

Updated `players`:

```
    Player #0: U2H U4D UJD UAD U9D
    Player #1: U8S U8D U9S UAH U4C
    Player #2: UTD U7D UQD U9H UJS
    Player #3: UAC U5C UKD UTH UJH
```

Full round-by-round play:

```
Player #3 holds the 2C and will lead off the game.

Round #1:
   Player 3’s hand: 3D 2S 9C JC 2C AC KH TS 5C 8H KD TH JH   Played 2C
   Player 0’s hand: TC 7H QS 2D 5H QC 3C 5S 2H 4D JD AD 9D   Played TC
   Player 1’s hand: 6D AS 4S 7S 6H 8S 7C 8C 8D 9S QH AH 4C   Played 7C
   Player 2’s hand: 3H 5D KC 6S 4H 3S TD 7D 6C QD 9H KS JS   Played KC
Round score: 0 point(s)  Points assigned to player #2.
Round #2:
   Player 2’s hand: 3H 5D 6S 4H 3S TD 7D 6C QD 9H KS JS   Played 3H
   Player 3’s hand: 3D 2S 9C JC AC KH TS 5C 8H KD TH JH   Played KH
   Player 0’s hand: 7H QS 2D 5H QC 3C 5S 2H 4D JD AD 9D   Played 7H
   Player 1’s hand: 6D AS 4S 7S 6H 8S 8C 8D 9S QH AH 4C   Played 6H
Round score: 4 point(s)  Points assigned to player #3.

Round #3:
   Player 3’s hand: 3D 2S 9C JC AC TS 5C 8H KD TH JH   Played 3D
   Player 0’s hand: QS 2D 5H QC 3C 5S 2H 4D JD AD 9D   Played 2D
   Player 1’s hand: 6D AS 4S 7S 8S 8C 8D 9S QH AH 4C   Played 6D
   Player 2’s hand: 5D 6S 4H 3S TD 7D 6C QD 9H KS JS   Played 5D
Round score: 0 point(s)  Points assigned to player #1.

Round #4:
   Player 1’s hand: AS 4S 7S 8S 8C 8D 9S QH AH 4C   Played AS
   Player 2’s hand: 6S 4H 3S TD 7D 6C QD 9H KS JS   Played 6S
   Player 3’s hand: 2S 9C JC AC TS 5C 8H KD TH JH   Played 2S
   Player 0’s hand: QS 5H QC 3C 5S 2H 4D JD AD 9D   Played QS
Round score: 13 point(s)  Points assigned to player #1.

Round #5:
   Player 1’s hand: 4S 7S 8S 8C 8D 9S QH AH 4C   Played 4S
   Player 2’s hand: 4H 3S TD 7D 6C QD 9H KS JS   Played 3S
   Player 3’s hand: 9C JC AC TS 5C 8H KD TH JH   Played TS
   Player 0’s hand: 5H QC 3C 5S 2H 4D JD AD 9D   Played 5S
Round score: 0 point(s)  Points assigned to player #3.

Round #6:
   Player 3’s hand: 9C JC AC 5C 8H KD TH JH   Played 9C
   Player 0’s hand: 5H QC 3C 2H 4D JD AD 9D   Played QC
   Player 1’s hand: 7S 8S 8C 8D 9S QH AH 4C   Played 8C
   Player 2’s hand: 4H TD 7D 6C QD 9H KS JS   Played 6C
Round score: 0 point(s)  Points assigned to player #0.

Round #7:
   Player 0’s hand: 5H 3C 2H 4D JD AD 9D   Played 5H
   Player 1’s hand: 7S 8S 8D 9S QH AH 4C   Played QH
   Player 2’s hand: 4H TD 7D QD 9H KS JS   Played 4H
   Player 3’s hand: JC AC 5C 8H KD TH JH   Played 8H
Round score: 4 point(s)  Points assigned to player #1.

Round #8:
   Player 1’s hand: 7S 8S 8D 9S AH 4C   Played 7S
   Player 2’s hand: TD 7D QD 9H KS JS   Played KS
   Player 3’s hand: JC AC 5C KD TH JH   Played JC
   Player 0's hand: 3C 2H 4D JD AD 9D   Played 3C
Round score: 0 point(s)  Points assigned to player #2.
```

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 `proj5.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 `proj5.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.