CSE 220: Systems Fundamentals I

Stony Brook University

Programming Assignment #5

Fall 2020

Assignment Due: Sunday, November 29, 2020 by 11:59 pm EST

Updates to this Document

- 11/10/2020: Correction to the return values in the Part 7 description. Highlighted in red.

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 implement functions that implement the MIPS assembly function calling conventions.
- Implement algorithms that process linked lists of structs.

Getting Started

Visit the course website and download the files hwk5.zip and MarsFall2020.jar. Fill in the following information at the top of hwk5.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 hwk5.asm you will find several function stubs that consist simply of `jr $ra` instructions. Your job
in this assignment is to implement all the functions as specified below. Do not change the function names, as the grading scripts will be looking for functions of the given names. However, you may implement additional helper functions of your own, but they must be saved in hwk5.asm. Helper functions will not necessarily be graded, but might be spot-checked to ensure you are following the MIPS function calling conventions.

If you are having difficulty implementing these functions, write out pseudocode or implement the functions in a higher-level language first. Once you understand the algorithm and what steps to perform, then translate the logic into MIPS assembly code.

Be sure to initialize all of your values (e.g., registers) within your functions. Never assume registers or memory will hold any particular values (e.g., zero). MARS initializes all of the registers and bytes of main memory to zeroes. The grading scripts will fill the registers and/or main memory with random values before calling your functions.

Finally, do not define a `.data` section in your hwk5.asm file. A submission that contains a `.data` section will probably result in a score of zero.

Important Information about CSE 220 Programming Projects

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

How Your CSE 220 Assignments Will Be Graded

With few exceptions, all aspects of your programming assignments will be graded entirely through automated means. Grading scripts will execute your code with input values (e.g., command-line arguments, function arguments) and will check for expected results (e.g., print-outs, return values, etc.) For this homework assignment you will be writing functions in assembly language. The functions will be tested independently of each other. This is very important to note, as you must take care that no function you write ever has side-effects or requires that other functions be called before the function in question is called. Both of these are generally considered bad practice in programming.

Some other items you should be aware of:

- Each test case must execute in 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.) will impact grading. Do not leave erroneous print-outs in your code.
- We will provide you with a small set of test cases for each assignment to give you a sense of how your work will be graded. It is your responsibility to test your code thoroughly by creating your own test cases.
- The testing framework we use for grading your work will not be released, but the test cases and expected results used for testing will be released.

Register Conventions

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

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

The following practices will result in loss of credit:

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

- “Hiding” or storing values in the $k, $f and $at registers or storing values in main memory by way of offsets to $gp. This is basically cheating or at best, a form of laziness, so don’t do it. We will comment out any such code we find.

How to Test Your Functions

To test your implemented functions, open the provided “main” files in MARS. Next, assemble the main file and run it. MARS will include the contents of any .asm files referenced with the .include directive(s) at the end of the file and then append the contents of your hwk5.asm file before assembling the program.

Each main file calls a single function with one of the sample test cases and prints any return value(s). You will need to change the arguments passed to the functions to test your functions with the other cases. To test each of your functions thoroughly, create your own test cases in those main files. Your submission will not be graded using the examples provided in this document or using the provided main file(s). Do not submit your main files with your hwk5.asm file – we will delete them. Also please note that these testing main files will not be used in grading.

Again, any modifications to the main files will not be graded. You will submit only your hwk5.asm for grading. Make sure that all code required for implementing your functions is included in the hwk5.asm file. To make sure that your code is self-contained, try assembling your hwk5.asm file by itself in MARS. If you get any errors (such as a missing label), this means that your hwk5.asm file is attempting to reference labels in the main file, which will not have the same names during grading!

A Final Reminder on How Your Work Will be Graded

It is imperative (crucial, essential, necessary, critically important) that you implement the functions below exactly as specified. Do not deviate from the specifications, even if you think you are implementing the program in a better way. Modify the contents of memory only as described in the function specifications!

Preliminaries

For this assignment you will be implementing a data structure containing linked lists to support an implementation of a variant of single-suit *Spider Solitaire*.

Data Structures

The assignment relies on two data structures given as structs:

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

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

Recall that the * notation indicates a *pointer*, or the address of a piece of data.

The num field in a Card struct encodes a playing card as follows:

- byte #0: the ASCII code of the card's rank: '0', '1', '2', …, '9'
- byte #1: the ASCII code of the card's suit: 'S', 'D', 'C' or 'H'
- byte #2: the ASCII code of 'u' or 'd', indicating whether the card is face-up or face down (we call this the card's "side")
- byte #3: the null-terminator

For example, "5Du" would be represented by the decimal value 7685173, or 0x00754435 in hexadecimal. 0x35 is the hexadecimal ASCII code for '5', 0x44 is the hexadecimal ASCII code for 'D', and 0x75 is the hexadecimal ASCII code for 'u'.

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

For example, a list of four cards might look like this:

```
card_list:
.word 4     # list's size
.word nodeA # address of list's head
nodeB:
.word 6574899
.word nodeC
nodeD:
.word 7689017
.word 0     # null pointer
nodeC:
.word 6574904
.word nodeD
nodeA:
.word 6574899
.word nodeB
```

Note that the nodes of the list are not necessarily laid out in order, or even contiguously, in memory.

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 CardNode, 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 Rules of *Spider Solitaire*

Spider Solitaire is a single player game played with two decks of cards. Go ahead, click the link and play a few rounds to familiarize yourself with the game. In this project, we will be implementing the classic **one-suit** version of *Spider Solitaire* in MIPS with a few minor alterations. Specifically, we will have 10 ranks (0♠ - 9♠) instead of 13 (2♠ - 10♠, J♠ - A♠), 80 total cards instead of 104, and 9 columns in the board instead of 10.

Objective:
The player's objective is to build full, descending sequences of cards within the columns. Once a ***full straight*** (9♠ - 8♠ - … - 2♠ - 1♠ - 0♠) is formed, all 10 of those cards are removed from play. Once all 80 cards are removed from play, the game is won.

Layout:
The game begins with a 9-column empty board and an 80-card, ♠-only deck containing 8 of each rank.

Play begins by dealing 44 face-down cards from the deck to the columns of the board. The leftmost 8 columns will have 5 cards, while the remaining rightmost column will have 4. Then, the top-most card in each column will be turned face-up.

The rest of the deck can be dealt out throughout the game at the player's discretion as a special move called the "deal-move". More on this in a bit.

Moving:
The player may remove the top card of a column (the "donor" column) and place it on the topmost card in another column (the "recipient" column) if it creates or continues a descending sequence. For

example, a 4♠ may be moved on top of a 5♠, but not vice-versa. As a result, a 9♠ may not be moved on top of *any* card.

A descending sequence of cards may also be moved as a package. For example, if a column ends with 8♠ - 5♠ - 4♠ - 3♠, the latter three cards may be moved as a pack to a 6♠ at the top of another column. Insertions are never allowed; moves will always take place at the tops of columns.

If a column happens to be empty, then it may be filled by any descending sequence of cards that can be moved.

A special move called the “deal-move” is used to spice things up when the player would like to add more cards to the board. When a “deal-move” is played, 9 cards from the deck are dealt face-up to each column. Since gameplay begins with 36 cards in the deck, over the course of a full game four “deal-moves” will be played.

There’s one twist, however: a “deal-move” can only be played when all of the columns contain at least one card. If you’re wondering why this is a rule, then that makes two of us.

After a move, a full straight in any column is removed if present. In addition, if the topmost card of any column is face-down, then it is turned face-up.

Part 1: Initialize an Empty `CardList` Struct

```
void init_list(CardList* card_list)
```

The `init_list` function takes 8 bytes of raw memory represented by `card_list` and initializes an empty CardList struct in that buffer by setting the list’s `size` and `head` fields to 0.

The function takes one argument:

- `card_list`: the starting address of a word-aligned, 8-byte buffer of uninitialized memory

Additional requirements:

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

Part 2: Append a Card to a `CardList`

```
int append_card(CardList* card_list, int card_num)
```

The `append_card` function takes a number `card_num` that represents a valid playing card and appends it to the end of the initialized list referenced by `card_list`. To accomplish this task, the function must use system call #9 to allocate memory for a `CardNode` object, set its fields as needed, and then append it to the end of `card_list` by linking it into the current end of the list. Don’t forget to adjust the `size` field of the list.

The function takes the following arguments, in this order:

- `card_list`: the address of a valid `CardList` struct, which might be empty.
- `card_num`: an integer that encodes a playing card. You may assume that `card_num` represents a valid playing card.

Returns in $v0:

- The size of `card_list` after appending the new card to the list.

Additional requirements:

- The function must not make any changes to main memory except as needed to create the `CardNode` object and link it to `card_list`.

Example #1: Append an item to an empty list.
```
card_list = empty
card_num = 6570802
```
Return value: 1
Updated `card_list`, after function call:
```
size: 1
contents: 6570802
```

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

```
card_list = 6574898 7685168 6572086 7684917 7689011
card_num = 6570802
```
Return value: 6
Updated `card_list`, after function call:
```
size: 6
contents: 6574898 7685168 6572086 7684917 7689011 6570802
```

Part 3: Initialize a Deck of Cards in Sorted Order

```
CardList* create_deck()
```

The `create_deck` function creates a new `CardList` consisting of the 80 playing cards in the prescribed order below and returns a pointer to the new list. Specifically, the list must contain 80 individual `CardNode` structs that represent cards in the following order (head node given first):

```
0Sd 1Sd 2Sd 3Sd 4Sd 5Sd 6Sd 7Sd 8Sd 9Sd 0Sd 1Sd 2Sd 3Sd 4Sd 5Sd 6Sd 7Sd
8Sd 9Sd 0Sd 1Sd 2Sd 3Sd 4Sd 5Sd 6Sd 7Sd 8Sd 9Sd 0Sd 1Sd 2Sd 3Sd 4Sd 5Sd
6Sd 7Sd 8Sd 9Sd 0Sd 1Sd 2Sd 3Sd 4Sd 5Sd 6Sd 7Sd 8Sd 9Sd 0Sd 1Sd 2Sd 3Sd
4Sd 5Sd 6Sd 7Sd 8Sd 9Sd 0Sd 1Sd 2Sd 3Sd 4Sd 5Sd 6Sd 7Sd 8Sd 9Sd 0Sd 1Sd
2Sd 3Sd 4Sd 5Sd 6Sd 7Sd 8Sd 9Sd
```

When encoded as unsigned integers, these cards are represented by the following hexadecimal values:

```
0x00645330 0x00645331 0x00645332 0x00645333 0x00645334 0x00645335
0x00645336 0x00645337 0x00645338 0x00645339 0x00645330 0x00645331
0x00645332 0x00645333 0x00645334 0x00645335 0x00645336 0x00645337
0x00645338 0x00645339 0x00645330 0x00645331 0x00645332 0x00645333
0x00645334 0x00645335 0x00645336 0x00645337 0x00645338 0x00645339
0x00645330 0x00645331 0x00645332 0x00645333 0x00645334 0x00645335
0x00645336 0x00645337 0x00645338 0x00645339 0x00645330 0x00645331
0x00645332 0x00645333 0x00645334 0x00645335 0x00645336 0x00645337
0x00645338 0x00645339 0x00645330 0x00645331 0x00645332 0x00645333
0x00645334 0x00645335 0x00645336 0x00645337 0x00645338 0x00645339
0x00645330 0x00645331 0x00645332 0x00645333 0x00645334 0x00645335
0x00645336 0x00645337 0x00645338 0x00645339 0x00645330 0x00645331
0x00645332 0x00645333 0x00645334 0x00645335 0x00645336 0x00645337
0x00645338 0x00645339
```

The function takes no arguments. The function must allocate on the heap the 8 bytes needed to create the `CardList` struct.

Returns in $v0:

- a pointer to a `CardList` of `CardNode` objects as described above.

Additional requirements:

- The function must not make any changes to main memory except as necessary.
- The function must call `init_list` and `append_card`.

Part 4: Deal a Deck of Shuffled Cards

```
void deal_starting_cards(CardList* board[], CardList* deck)
```

The `deal_starting_cards` function takes an array of *pointers* to 9 empty `CardList` structs that represents the game board and a CardList of 80 face-down cards representing a shuffled deck of cards, and deals out 44 cards from the deck onto the game board. The first 35 cards are dealt face-down, one card at a time across the tops of the columns. The first card is laid on column #0, the second card on column #1, etc., until the ninth card is laid on column #8. The 10th card is laid on column #0, the 11th on column #1, etc., until the 35th card is laid on column #7 (not 8). Then, nine more cards are drawn from the deck and placed face-up on the columns of the board, starting with column #8 and wrapping around. In this manner, the rightmost column (#8) will have one fewer face-down card than the others. Don't forget to adjust the sizes of the board's columns and the deck as needed.

As an example, suppose we start with the deck of cards shown below:

```
6Sd 4Sd 1Sd 2Sd 8Sd 3Sd 4Sd 8Sd 8Sd 2Sd 3Sd 9Sd 0Sd 7Sd 9Sd 0Sd 1Sd 0Sd
0Sd 7Sd 7Sd 9Sd 5Sd 6Sd 8Sd 4Sd 3Sd 1Sd 7Sd 6Sd 8Sd 1Sd 2Sd 0Sd 4Sd 9Sd
8Sd 1Sd 8Sd 9Sd 6Sd 0Sd 6Sd 5Sd 7Sd 9Sd 1Sd 9Sd 4Sd 6Sd 5Sd 1Sd 7Sd 1Sd
5Sd 2Sd 6Sd 2Sd 8Sd 0Sd 3Sd 6Sd 5Sd 3Sd 9Sd 0Sd 5Sd 3Sd 4Sd 5Sd 5Sd 7Sd
3Sd 4Sd 7Sd 4Sd 3Sd 2Sd 2Sd 2Sd
```

The expected contents of the game board is depicted below (the suit is omitted from the visualization):

```
#0   #1   #2   #3   #4   #5   #6   #7   #8
--   --   --   --   --   --   --   --   --
6d   4d   1d   2d   8d   3d   4d   8d   8d    <-- bottoms of piles
2d   3d   9d   0d   7d   9d   0d   1d   0d
0d   7d   7d   9d   5d   6d   8d   4d   3d
1d   7d   6d   8d   1d   2d   0d   4d   9u
8u   1u   8u   9u   6u   0u   6u   5u         <-- tops of piles
```

Note that the top cards are all face-up.

The contents of the board (shown sideways) will be represented by the following linked lists.

```
Column #0: 0x00645336 0x00645332 0x00645330 0x00645331 0x00755338
Column #1: 0x00645334 0x00645333 0x00645337 0x00645337 0x00755331
Column #2: 0x00645331 0x00645339 0x00645337 0x00645336 0x00755338
Column #3: 0x00645332 0x00645330 0x00645339 0x00645338 0x00755339
Column #4: 0x00645338 0x00645337 0x00645335 0x00645331 0x00755336
Column #5: 0x00645333 0x00645339 0x00645336 0x00645332 0x00755330
Column #6: 0x00645334 0x00645330 0x00645338 0x00645330 0x00755336
Column #7: 0x00645338 0x00645331 0x00645334 0x00645334 0x00755335
Column #8: 0x00645338 0x00645330 0x00645333 0x00755339
```

After the deal, the deck will contain the following cards:

```
7Sd 9Sd 1Sd 9Sd 4Sd 6Sd 5Sd 1Sd 7Sd 1Sd 5Sd 2Sd 6Sd 2Sd 8Sd 0Sd 3Sd 6Sd
5Sd 3Sd 9Sd 0Sd 5Sd 3Sd 4Sd 5Sd 5Sd 7Sd 3Sd 4Sd 7Sd 4Sd 3Sd 2Sd 2Sd 2Sd
```

The argument `board` is not an array of 9 `CardList` structs, but rather an array of pointers to `CardList` structs. Thus, $a0 contains the starting address of this array of pointers. For example, `board[3]` stores the address of the `CardList` struct that stores the cards of column #3 in the game board. That `CardList`'s address is stored at the address `$a0 + 12`. We can load (`lw`) the value at that address (into say, $t0, via `lw $t0, 12($a0)`) and we now have the address of that CardList in $t0. (Pseudocode: `$t0 = board[3]`) The first 4 bytes of a `CardList` contain the list's size. So if we executed `lw $t1, 0($t0)`, we could get the list's length. (Pseudocode: `$t1 = board[3].size`) The code `lw $t1, 4($t0)` would give us the address of the `head` `CardNode` of the linked list.

The function takes the following arguments, in this order:

- `board`: an array of 9 pointers to initialized `CardList` structs (`size` and `head` in all lists are 0)
- `deck`: a `CardList` of exactly 80 face-down cards

Additional requirements:

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

Example:

Starting deck:

```
8Sd 4Sd 6Sd 6Sd 5Sd 5Sd 4Sd 3Sd 7Sd 7Sd 5Sd 9Sd 1Sd 3Sd 4Sd 8Sd 7Sd 0Sd
1Sd 1Sd 2Sd 1Sd 4Sd 8Sd 6Sd 9Sd 2Sd 2Sd 6Sd 0Sd 4Sd 2Sd 3Sd 8Sd 3Sd 4Sd
8Sd 6Sd 7Sd 7Sd 2Sd 3Sd 5Sd 1Sd 8Sd 9Sd 0Sd 0Sd 8Sd 6Sd 4Sd 0Sd 5Sd 0Sd
5Sd 1Sd 3Sd 5Sd 1Sd 7Sd 8Sd 1Sd 9Sd 6Sd 7Sd 0Sd 9Sd 9Sd 2Sd 2Sd 9Sd 7Sd
5Sd 3Sd 4Sd 6Sd 0Sd 2Sd 9Sd 3Sd
```

Expected game board:

```
#0   #1   #2   #3   #4   #5   #6   #7   #8
--   --   --   --   --   --   --   --   --
8d   4d   6d   6d   5d   5d   4d   3d   7d
7d   5d   9d   1d   3d   4d   8d   7d   0d
1d   1d   2d   1d   4d   8d   6d   9d   2d
2d   6d   0d   4d   2d   3d   8d   3d   4u
8u   6u   7u   7u   2u   3u   5u   1u
```

Contents of board's linked lists:

```
Column #0: 0x00645338 0x00645337 0x00645331 0x00645332 0x00755338
Column #1: 0x00645334 0x00645335 0x00645331 0x00645336 0x00755336
Column #2: 0x00645336 0x00645339 0x00645332 0x00645330 0x00755337
Column #3: 0x00645336 0x00645331 0x00645331 0x00645334 0x00755337
Column #4: 0x00645335 0x00645333 0x00645334 0x00645332 0x00755332
Column #5: 0x00645335 0x00645334 0x00645338 0x00645333 0x00755333
Column #6: 0x00645334 0x00645338 0x00645336 0x00645338 0x00755335
Column #7: 0x00645333 0x00645337 0x00645339 0x00645333 0x00755331
Column #8: 0x00645337 0x00645330 0x00645332 0x00755334
```

Ending deck:

```
8Sd 9Sd 0Sd 0Sd 8Sd 6Sd 4Sd 0Sd 5Sd 0Sd 5Sd 1Sd 3Sd 5Sd 1Sd 7Sd 8Sd 1Sd
9Sd 6Sd 7Sd 0Sd 9Sd 9Sd 2Sd 2Sd 9Sd 7Sd 5Sd 3Sd 4Sd 6Sd 0Sd 2Sd 9Sd 3Sd
```

Part 5: Get a Card from a CardList

```
int, int get_card(CardList* card_list, int index)
```

The `get_card` function retrieves the integer representing the card at the given position in the list (i.e., the value of the `CardNode.card` at position `index`). The list is 0-indexed. If the list is empty or `index` is invalid for the given list, the function returns -1, -1. Otherwise, in $v0, the function returns 1 if the card is face-down, or 2 if the card is face-up. In $v1, the function returns the number encoding the card itself. (You might ask why return a special code in $v0 if we can just get it from $v1? Answer: to provide a small courtesy to the caller function.)

The function takes the following arguments, in this order:

- `card_list`: a valid `CardList` of cards
- `index`: the position from which we want to retrieve a value in `card_list`

Returns in $v0:

- 1 if the card at index `index` is face-down
- 2 if the card at index `index` is face-up
- -1 if `index` is invalid

Returns in $v1:

- the number stored at index `index` of `card_list`
- -1 if `index` is invalid

Additional requirements:

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

Example #1: Attempt to get a card from an empty list.
```
card_list = empty
index = 0
```
Return values: -1, -1

Example #2: Illegal index.
```
card_list = 6574898 7685168 6572086 7684917 7689011
index = 5
```
Return values: -1, -1

Example #3: Get a card at index 0.
```
card_list = 6574898 7685168 6572086 7684917 7689011
index = 0
```
Return values: 1, 6574898

Example #4: Get a card in the middle of a list.
```
card_list = 6574898 7685168 6572086 7684917 7689011 6574898 7684912
6572083
index = 3
```
Return values: 2, 7684917

Example #5: Get the last card in the list.
```
card_list = 6574898 7685168 6572086 7684917 7689011 6574898 7684912
6572083
index = 7
```
Return values: 1, 6572083

Part 6: Determine if a Proposed Move is Valid

```
int check_move(CardList* board[], CardList* deck, int move)
```

The `check_move` function determines if the potential move encoded by the `move` argument is legal or illegal, returning an integer code that communicates the result. This function does not make any actual changes to the board, but merely indicates whether or not the move can be made. A move in the game is encoded by four bytes:

- byte #0: the column we want to move a card (or cards) from (a.k.a. the "donor" column)
- byte #1: the row of the card(s) we want to move from. For a move of a single card, this would just be the row of the card. For a move of multiple cards, this would be the row of the bottommost card in the pack
- byte #2: the column of the board we want to move the card(s) to (a.k.a. the "recipient" column)
- byte #3: 0 if we want to move one or more cards; 1 if we want to perform a deal-move

If byte #3 is 1, indicating a deal-move, the other three bytes must be all 0. For example, 0x00010205 encodes the action to move cards at and above column 5, row 2 to column 1 (remember that our MIPS simulator is little-endian).

The function takes the following arguments, in this order:

- `board`: an array of 9 pointers to pre-initialized `CardList` structs
- `deck`: a pointer to a valid `CardList`
- `move`: an integer that encodes a game move as described above

The various **error** conditions that are considered by the function **must be checked in the order listed below**. As an illegal move may have multiple errors, this is important to ensure that the function will return the value we are looking for. For example, the function will return -4 only if the error conditions above it (-1, -2, -3) are not satisfied. The **success** conditions **should also be checked in the order listed below**. But, it may be correct to check a success condition before some error conditions. The exact order is for you to figure out.

Returns in $v0:

For an **error**:

(Deal-move parameter error)

- -1 if the `move` argument encodes an invalid deal-move (i.e., byte #3 is 1 but the other three bytes are not all zeros) or the value in the deal-move byte is neither 0 nor 1

(Illegal deal-move error)

- -2 if the `move` argument is a valid deal-move, but the deck is empty or if at least one column of the board is empty

(Normal move parameter errors)

- -3 if the donor column or the recipient column encoded in the `move` argument is invalid
- -4 if the donor row encoded in the `move` argument is invalid
- -5 if the donor and recipient columns encoded in the `move` argument are valid but are the same number (moves from a column to itself doesn't count!)

(Illegal normal move errors)

- -6 if the card at the donor column and row encoded in the `move` argument is face-down
- -7 if the cards in the donor column that would be moved are not listed in descending, contiguous order (i.e, there can be no gaps; the cards 6♠ - 5♠ - 3♠ - 2♠ would not be legal to move as a unit)
- -8 if the recipient column is not empty, and the rank of the selected card is not one less than the rank of the top card in the recipient column

  For a **success**:

- 1 if the move is a legal deal-move
- 2 if the move is a legal move (but not a deal-move) and the recipient column is empty
- 3 if the move is a legal move (but not a deal-move) and the recipient column is non-empty

Additional requirements:

- The function must not make any changes to main memory.
- The function must call `get_card`.

In the examples below, notation like `[n, m]` means column #n, row #m where row #0 is where the list heads lie.

Example #1: Deal-move

```
deck = 9Sd 9Sd 4Sd 8Sd 3Sd 8Sd 1Sd 0Sd 8Sd 3Sd 7Sd 8Sd 6Sd 7Sd 9Sd 1Sd 2Sd
9Sd 4Sd 5Sd 3Sd 2Sd 6Sd 5Sd 4Sd 0Sd 6Sd 9Sd 4Sd 3Sd 0Sd 5Sd 8Sd 0Sd 4Sd
1Sd
```

`move = 0x01000000` (Deal-move)

Given board:

```
#0   #1   #2   #3   #4   #5   #6   #7   #8
--   --   --   --   --   --   --   --   --
4d   6d   0d   6d   3d   1d   0d   2d   5d   <-- row #0
5d   4d   3d   3d   7d   0d   6d   5d   8d   <-- row #1
7d   1d   6d   5d   1d   7d   0d   7d   2d       etc.
1d   2d   9d   7d   5d   6d   3d   1d   8u
2u   7u   9u   4u   9u   2u   2u   8u
```

Return value in $v0: 1

Example #2: Legal move of cards to an empty column

```
deck = 9Sd 9Sd 4Sd 8Sd 3Sd 8Sd 1Sd 0Sd 8Sd 3Sd 7Sd 8Sd 6Sd 7Sd 9Sd 1Sd 2Sd
9Sd 4Sd 5Sd 3Sd 2Sd 6Sd 5Sd 4Sd 0Sd 6Sd 9Sd 4Sd 3Sd 0Sd 5Sd 8Sd 0Sd 4Sd
1Sd
```

`move = 0x00010205` (Move cards at and above [5, 2] to column 1)
Given board:
```
#0   #1   #2   #3   #4   #5   #6   #7   #8
--   --   --   --   --   --   --   --   --
4d        0d   6d   3d   1d   0d   2d   5d
5d        3d   3d   7d   0d   6d   5d   8d
7d        6d   5d   1d   7u   0d   7d   2d
1d        9d   7d   5d   6u   3d   1d   8u
2u        9u   4u   9u   5u   2u   8u
```
Return value in $v0: 2

Example #3: Legal move of one card to an empty column
```
deck = 4Sd 6Sd 0Sd 6Sd 3Sd 1Sd 0Sd 2Sd 5Sd 5Sd 4Sd 3Sd 3Sd 7Sd 0Sd 6Sd 5Sd
8Sd 7Sd 1Sd 6Sd 5Sd 1Sd 7Sd 0Sd 7Sd 2Sd 1Sd 2Sd 9Sd 7Sd 5Sd 6Sd 3Sd 1Sd
8Sd 2Sd 7Sd 9Sd 4Sd 9Sd 2Sd 2Sd 8Sd 9Sd 9Sd 4Sd 8Sd 3Sd 8Sd 1Sd 0Sd 8Sd
3Sd 7Sd 8Sd 6Sd 7Sd 9Sd 1Sd 2Sd 9Sd 4Sd 5Sd 3Sd 2Sd 6Sd 5Sd 4Sd 0Sd 6Sd
9Sd 4Sd 3Sd 0Sd 5Sd 8Sd 0Sd 4Sd 1Sd
```
`move = 0x00010403` (Move cards at and above [3, 4] to column 1)
Given board:
```
#0   #1   #2   #3   #4   #5   #6   #7   #8
--   --   --   --   --   --   --   --   --
4d        0d   6d   3d   1d   0d   2d   5d
5d        3d   3d   7d   0d   6d   5d   8d
7d        6d   5d   1d   7u   0d   7d   2d
1d        9d   7d   5d   6u   3d   1d   8u
2u        9u   4u   9u   5u   2u   8u
```
Return value in $v0: 2

Example #4: Legal move of several cards to a non-empty column
```
deck = 9Sd 9Sd 4Sd 8Sd 3Sd 8Sd 1Sd 0Sd 8Sd 3Sd 7Sd 8Sd 6Sd 7Sd 9Sd 1Sd 2Sd
9Sd 4Sd 5Sd 3Sd 2Sd 6Sd 5Sd 4Sd 0Sd 6Sd 9Sd 4Sd 3Sd 0Sd 5Sd 8Sd 0Sd 4Sd
1Sd
```
`move = 0x00030306` (Move cards at and above [6, 3] to column 3)
Given board:
```
#0   #1   #2   #3   #4   #5   #6   #7   #8
--   --   --   --   --   --   --   --   --
4d   6d   0d   6d   3d   1d   0d   2d   5d
5d   4d   3d   3d   7d   0d   6d   5d   8d
7d   1d   6d   5d   1d   7d   0d   7d   2d
1d   2d   9d   7d   5d   6d   3u   1d   8u
2u   7u   9u   4u   9u   2u   2u   8u
```
Return value in $v0: 3

Example #5: Invalid move encoded
```
deck = empty
```

`move = 0x06030102` (Illegal move)
Given board:
```
#0   #1   #2   #3   #4   #5   #6   #7   #8
--   --   --   --   --   --   --   --   --
4d   6d   0d   6d   3d   1d   0d   2d   5d
5d   4d   3d   3d   7d   0d   6d   5d   8d
7d   1d   6d   5d   1d   7d   0d   7d   2d
1d   2d   9d   7d   5d   6d   3u   1d   8u
2u   7u   9u   4u   9u   2u   2u   8u
```
Return value in $v0: -1

Example #6: Attempted deal-move with an empty deck
`deck = empty`
`move = 0x01000000` (Deal-move)
Given board:
```
#0   #1   #2   #3   #4   #5   #6   #7   #8
--   --   --   --   --   --   --   --   --
4d   6d   0d   6d   3d   1d   0d   2d   5d
5d   4d   3d   3d   7d   0d   6d   5d   8d
7d   1d   6d   5d   1d   7d   0d   7d   2d
1d   2d   9d   7d   5d   6d   3u   1d   8u
2u   7u   9u   4u   9u   2u   2u   8u
```
Return value in $v0: -2

Example #7: Invalid column number
```
deck = 4Sd 6Sd 0Sd 6Sd 3Sd 1Sd 0Sd 2Sd 5Sd 5Sd 4Sd 3Sd 3Sd 7Sd 0Sd 6Sd 5Sd
8Sd 7Sd 1Sd 6Sd 5Sd 1Sd 7Sd 0Sd 7Sd 2Sd 1Sd 2Sd 9Sd 7Sd 5Sd 6Sd 3Sd 1Sd
8Sd 2Sd 7Sd 9Sd 4Sd 9Sd 2Sd 2Sd 8Sd 9Sd 9Sd 4Sd 8Sd 3Sd 8Sd 1Sd 0Sd 8Sd
3Sd 7Sd 8Sd 6Sd 7Sd 9Sd 1Sd 2Sd 9Sd 4Sd 5Sd 3Sd 2Sd 6Sd 5Sd 4Sd 0Sd 6Sd
9Sd 4Sd 3Sd 0Sd 5Sd 8Sd 0Sd 4Sd 1Sd
```
`move = 0x00110401` (Move cards at and above [1, 4] to column 17)
Given board:
```
#0   #1   #2   #3   #4   #5   #6   #7   #8
--   --   --   --   --   --   --   --   --
4d   6d   0d   6d   3d   1d   0d   2d   5d
5d   4d   3d   3d   7d   0d   6d   5d   8d
7d   1d   6d   5d   1d   7d   0d   7d   2d
1d   2d   9d   7d   5d   6d   3u   1d   8u
2u   7u   9u   4u   9u   2u   2u   8u
```
Return value in $v0: -3

Example #8: Invalid row number
```
deck = 4Sd 6Sd 0Sd 6Sd 3Sd 1Sd 0Sd 2Sd 5Sd 5Sd 4Sd 3Sd 3Sd 7Sd 0Sd 6Sd 5Sd
8Sd 7Sd 1Sd 6Sd 5Sd 1Sd 7Sd 0Sd 7Sd 2Sd 1Sd 2Sd 9Sd 7Sd 5Sd 6Sd 3Sd 1Sd
8Sd 2Sd 7Sd 9Sd 4Sd 9Sd 2Sd 2Sd 8Sd 9Sd 9Sd 4Sd 8Sd 3Sd 8Sd 1Sd 0Sd 8Sd
```

```
3Sd 7Sd 8Sd 6Sd 7Sd 9Sd 1Sd 2Sd 9Sd 4Sd 5Sd 3Sd 2Sd 6Sd 5Sd 4Sd 0Sd 6Sd
9Sd 4Sd 3Sd 0Sd 5Sd 8Sd 0Sd 4Sd 1Sd
```
`move = 0x00040701` (Move cards at and above [1, 7] to column 4)
Given board:
```
#0   #1   #2   #3   #4   #5   #6   #7   #8
--   --   --   --   --   --   --   --   --
4d   6d   0d   6d   3d   1d   0d   2d   5d
5d   4d   3d   3d   7d   0d   6d   5d   8d
7d   1d   6d   5d   1d   7d   0d   7d   2d
1d   2d   9d   7d   5d   6d   3u   1d   8u
2u   7u   9u   4u   9u   2u   2u   8u
```
Return value in $v0: -4

Example #9: Donor and recipient columns are the same
```
deck = 4Sd 6Sd 0Sd 6Sd 3Sd 1Sd 0Sd 2Sd 5Sd 5Sd 4Sd 3Sd 3Sd 7Sd 0Sd 6Sd 5Sd
8Sd 7Sd 1Sd 6Sd 5Sd 1Sd 7Sd 0Sd 7Sd 2Sd 1Sd 2Sd 9Sd 7Sd 5Sd 6Sd 3Sd 1Sd
8Sd 2Sd 7Sd 9Sd 4Sd 9Sd 2Sd 2Sd 8Sd 9Sd 9Sd 4Sd 8Sd 3Sd 8Sd 1Sd 0Sd 8Sd
3Sd 7Sd 8Sd 6Sd 7Sd 9Sd 1Sd 2Sd 9Sd 4Sd 5Sd 3Sd 2Sd 6Sd 5Sd 4Sd 0Sd 6Sd
9Sd 4Sd 3Sd 0Sd 5Sd 8Sd 0Sd 4Sd 1Sd
```
`move = 0x00010401` (Move cards at and above [1, 4] to column 1)
Given board:
```
#0   #1   #2   #3   #4   #5   #6   #7   #8
--   --   --   --   --   --   --   --   --
4d   6d   0d   6d   3d   1d   0d   2d   5d
5d   4d   3d   3d   7d   0d   6d   5d   8d
7d   1d   6d   5d   1d   7d   0d   7d   2d
1d   2d   9d   7d   5d   6d   3u   1d   8u
2u   7u   9u   4u   9u   2u   2u   8u
```
Return value in $v0: -5

Example #10: Attempt to move a face-down card
```
deck = 4Sd 6Sd 0Sd 6Sd 3Sd 1Sd 0Sd 2Sd 5Sd 5Sd 4Sd 3Sd 3Sd 7Sd 0Sd 6Sd 5Sd
8Sd 7Sd 1Sd 6Sd 5Sd 1Sd 7Sd 0Sd 7Sd 2Sd 1Sd 2Sd 9Sd 7Sd 5Sd 6Sd 3Sd 1Sd
8Sd 2Sd 7Sd 9Sd 4Sd 9Sd 2Sd 2Sd 8Sd 9Sd 9Sd 4Sd 8Sd 3Sd 8Sd 1Sd 0Sd 8Sd
3Sd 7Sd 8Sd 6Sd 7Sd 9Sd 1Sd 2Sd 9Sd 4Sd 5Sd 3Sd 2Sd 6Sd 5Sd 4Sd 0Sd 6Sd
9Sd 4Sd 3Sd 0Sd 5Sd 8Sd 0Sd 4Sd 1Sd
```
`move = 0x00070201` (Move cards at and above [1, 2] to column 7)
Given board:
```
#0   #1   #2   #3   #4   #5   #6   #7   #8
--   --   --   --   --   --   --   --   --
4d   6d   0d   6d   3d   1d   0d   2d   5d
5d   4d   3d   3d   7d   0d   6d   5d   8d
7d   1d   6d   5d   1d   7d   0d   7d   2d
1d   2d   9d   7d   5d   6d   3u   1d   8u
2u   7u   9u   4u   9u   2u   2u   8u
```

Return value in $v0: -6

Example #11: Attempt to move cards that are not in descending order

```
deck = 9Sd 9Sd 4Sd 8Sd 3Sd 8Sd 1Sd 0Sd 8Sd 3Sd 7Sd 8Sd 6Sd 7Sd 9Sd 1Sd 2Sd
9Sd 4Sd 5Sd 3Sd 2Sd 6Sd 5Sd 4Sd 0Sd 6Sd 9Sd 4Sd 3Sd 0Sd 5Sd 8Sd 0Sd 4Sd
1Sd
```

`move = 0x00080205` (Move cards at and above [5, 2] to column 8)

Given board:

```
#0   #1   #2   #3   #4   #5   #6   #7   #8
--   --   --   --   --   --   --   --   --
4d   6d   0d   6d   3d   1d   0d   2d   5d
5d   4d   3d   3d   7d   0d   6d   5d   8d
7d   1d   6d   5d   1d   7u   0d   7d   2d
1d   2d   9d   7d   5d   6u   3d   1d   8u
2u   7u   9u   4u   9u   2u   2u   8u
```

Return value in $v0: -7

Example #12: Attempt to move movable cards to an invalid recipient column

```
deck = 9Sd 9Sd 4Sd 8Sd 3Sd 8Sd 1Sd 0Sd 8Sd 3Sd 7Sd 8Sd 6Sd 7Sd 9Sd 1Sd 2Sd
9Sd 4Sd 5Sd 3Sd 2Sd 6Sd 5Sd 4Sd 0Sd 6Sd 9Sd 4Sd 3Sd 0Sd 5Sd 8Sd 0Sd 4Sd
1Sd
```

`move = 0x00030205` (Move cards at and above [5, 2] to column 3)

Given board:

```
#0   #1   #2   #3   #4   #5   #6   #7   #8
--   --   --   --   --   --   --   --   --
4d   6d   0d   6d   3d   1d   0d   2d   5d
5d   4d   3d   3d   7d   0d   6d   5d   8d
7d   1d   6d   5d   1d   7u   0d   7d   2d
1d   2d   9d   7d   5d   6u   3d   1d   8u
2u   7u   9u   4u   9u   5u   2u   8u
```

Return value in $v0: -8

Part 7: Clear a Full Straight of Cards

```
int clear_full_straight(CardList* board[], int col_num)
```

The `clear_full_straight` function attempts to clear out the top 10 cards of board[col_num], provided that those 10 cards are all face-up and represent a full-straight (i.e., face-up cards 9♠ - 8♠ - … - 1♠ - 0♠). When a full straight is removed from the board, the new top card of the column is flipped to be face-up. Don't forget to adjust the size of the modified column as needed.

The function takes the following arguments, in this order:

- `board`: an array of 9 pointers to initialized `CardList` structs
- `col_num`: the 0-based column number in which to check for the full straight

Returns in $v0:

- -1 if `col_num` is invalid
- -2 if `board[col_num]` contains fewer than 10 cards
- -3 if `board[col_num]` contains at least 10 cards but no full straight can be cleared from the column
- **2** if a full straight is cleared out of `board[col_num]`, leaving the column empty of cards
- **1** if a full straight is cleared out of `board[col_num]`, leaving at least one card behind in the column

Additional requirements:

- The function must not make any changes to main memory except when necessary.
- The function must call `get_card` at least once in a manner that makes logical sense.

Example #1: Invalid column number.
Given `board`:

```
#0   #1   #2   #3   #4   #5   #6   #7   #8
--   --   --   --   --   --   --   --   --
6d   4d   1d   2d   8d   3d   4d   8d   8d
2d   3d   9d   0d   7d   9d   0d   1d   0d
0d   7d   7d   9d   5d   6d   8d   4d   3d
1d   7d   6d   8d   1d   2d   0d   4d   9u
8u   1u   8u   9u   6u   0u   6u   5u
col_num = 10
```

Return value in $v0: -1
Expected `board`:

```
#0   #1   #2   #3   #4   #5   #6   #7   #8
--   --   --   --   --   --   --   --   --
6d   4d   1d   2d   8d   3d   4d   8d   8d
2d   3d   9d   0d   7d   9d   0d   1d   0d
0d   7d   7d   9d   5d   6d   8d   4d   3d
1d   7d   6d   8d   1d   2d   0d   4d   9u
8u   1u   8u   9u   6u   0u   6u   5u
```

Example #2: Check for straight in a column that is too short.
Given `board`:

```
#0   #1   #2   #3   #4   #5   #6   #7   #8
--   --   --   --   --   --   --   --   --
6d   4d   1d   2d   8d   3d   4d   8d   8d
2d   3d   9d   0d   7d   9d   0d   1d   0d
0d   7d   7d   9d   5d   6d   8d   4d   3d
1d   7d   6d   8d   1d   2d   0d   4d   9u
8u   1u   8u   9u   6u   0u   6u   5u
col_num = 2
```

Return value in $v0: -2

Expected `board`:

```
#0   #1   #2   #3   #4   #5   #6   #7   #8
--   --   --   --   --   --   --   --   --
6d   4d   1d   2d   8d   3d   4d   8d   8d
2d   3d   9d   0d   7d   9d   0d   1d   0d
0d   7d   7d   9d   5d   6d   8d   4d   3d
1d   7d   6d   8d   1d   2d   0d   4d   9u
8u   1u   8u   9u   6u   0u   6u   5u
```

Example #3: Column contains a straight, but there are some additional cards above.
Given `board`:

```
#0   #1   #2   #3   #4   #5   #6   #7   #8
--   --   --   --   --   --   --   --   --
9u
8u
7u
6u
5u
4u
3u
2u
1u
0u
2u
0u
6u
col_num = 0
```

Return value in $v0: -3
Expected `board`:

```
#0   #1   #2   #3   #4   #5   #6   #7   #8
--   --   --   --   --   --   --   --   --
9u
8u
7u
6u
5u
4u
3u
2u
1u
0u
2u
0u
6u
```

Example #4: Column contains a partially face-down straight.
Given `board`:
```
#0   #1   #2   #3   #4   #5   #6   #7   #8
--   --   --   --   --   --   --   --   --
                    9d
                    8d
                    7d
                    6u
                    5u
                    4u
                    3u
                    2u
                    1u
                    0u
col_num = 4
```
Return value in $v0: -3
Expected `board`:
```
#0   #1   #2   #3   #4   #5   #6   #7   #8
--   --   --   --   --   --   --   --   --
                    9d
                    8d
                    7d
                    6u
                    5u
                    4u
                    3u
                    2u
                    1u
                    0u
```

Example #5: Column contains a straight and some additional cards below.
Given `board`:
```
#0   #1   #2   #3   #4   #5   #6   #7   #8
--   --   --   --   --   --   --   --   --
2d
0d
6u
9u
8u
7u
6u
5u
4u
3u
2u
```

```
1u
0u
col_num = 0
```
Return value in $v0: 1

Expected `board`:
```
#0   #1   #2   #3   #4   #5   #6   #7   #8
--   --   --   --   --   --   --   --   --
2d
0d
6u
```

Example #6: Column contains a straight with no additional cards above.

Given `board`:
```
#0   #1   #2   #3   #4   #5   #6   #7   #8
--   --   --   --   --   --   --   --   --
                    9u
                    8u
                    7u
                    6u
                    5u
                    4u
                    3u
                    2u
                    1u
                    0u
col_num = 4
```
Return value in $v0: 2

Expected `board`:
```
#0   #1   #2   #3   #4   #5   #6   #7   #8
--   --   --   --   --   --   --   --   --
```
(empty board)

Part 8: Perform a Valid Deal-move

```
void deal_move(CardList* board[], CardList* deck)
```

The `deal_move` function performs a deal-move, drawing 9 cards from the deck and laying them face-up on the top of each of the 9 columns of the game board. This function does not need to perform any validation. For example, it does not need to check if any of the board's columns are empty or if the deck is empty. Don't forget to adjust the sizes of the board's columns and the deck as needed.

The function takes the following arguments, in this order:

- `board`: an array of 9 pointers to initialized `CardList` structs
- `deck`: a pointer to a valid `CardList`

Additional requirements:

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

Example #1: Deal-move for a starting configuration of cards.
Given deck: `2Sd 3Sd 8Sd 7Sd 0Sd 5Sd 6Sd 8Sd 6Sd 7Sd 0Sd 1Sd 3Sd 2Sd 8Sd 0Sd 6Sd 0Sd 6Sd 0Sd 7Sd 1Sd 3Sd 9Sd 3Sd 3Sd 4Sd 5Sd 4Sd 5Sd 4Sd 0Sd 1Sd 1Sd 9Sd 9Sd`
Given board:

```
#0   #1   #2   #3   #4   #5   #6   #7   #8
--   --   --   --   --   --   --   --   --
6d   7d   3d   5d   4d   8d   4d   2d   6d
9d   1d   0d   9d   2d   5d   2d   4d   8d
2d   9d   3d   1d   8d   1d   9d   0d   4d
5d   5d   8d   5d   7d   9d   4d   8d   2u
7u   7u   7u   2u   3u   1u   6u   6u
```

Expected deck: `7Sd 0Sd 1Sd 3Sd 2Sd 8Sd 0Sd 6Sd 0Sd 6Sd 0Sd 7Sd 1Sd 3Sd 9Sd 3Sd 3Sd 4Sd 5Sd 4Sd 5Sd 4Sd 0Sd 1Sd 1Sd 9Sd 9Sd`
Expected board:

```
#0   #1   #2   #3   #4   #5   #6   #7   #8
--   --   --   --   --   --   --   --   --
6d   7d   3d   5d   4d   8d   4d   2d   6d
9d   1d   0d   9d   2d   5d   2d   4d   8d
2d   9d   3d   1d   8d   1d   9d   0d   4d
5d   5d   8d   5d   7d   9d   4d   8d   2u
7u   7u   7u   2u   3u   1u   6u   6u   6u
2u   3u   8u   7u   0u   5u   6u   8u
```

Example #2: Deal-move for a game in-progress #1.
Given deck: `5Sd 1Sd 0Sd 4Sd 5Sd 6Sd 4Sd 8Sd 4Sd 7Sd 2Sd 6Sd 0Sd 9Sd 7Sd 3Sd 6Sd 0Sd 4Sd 6Sd 0Sd 6Sd 8Sd 3Sd 9Sd 0Sd 2Sd 8Sd 9Sd 5Sd 1Sd 4Sd 4Sd 8Sd 5Sd 7Sd`
Given board:

```
#0   #1   #2   #3   #4   #5   #6   #7   #8
--   --   --   --   --   --   --   --   --
8d   9d   4d   5d   9d   3d   0d   2d   1d
3d   3d   8d   5d   1u   3u   7d   2d   0d
4u   7d   6d   8u             7d   0d   2d
     1d   1d                  9u   2d   1u
     7u   2u                       3u
```

Expected deck: `7Sd 2Sd 6Sd 0Sd 9Sd 7Sd 3Sd 6Sd 0Sd 4Sd 6Sd 0Sd 6Sd 8Sd 3Sd 9Sd 0Sd 2Sd 8Sd 9Sd 5Sd 1Sd 4Sd 4Sd 8Sd 5Sd 7Sd`
Expected board:

```
#0   #1   #2   #3   #4   #5   #6   #7   #8
```

```
--   --   --   --   --   --   --   --   --
8d   9d   4d   5d   9d   3d   0d   2d   1d
3d   3d   8d   5d   1u   3u   7d   2d   0d
4u   7d   6d   8u   5u   6u   7d   0d   2d
5u   1d   1d   4u             9u   2d   1u
     7u   2u                  4u   3u   4u
     1u   0u                       8u
```

Example #3: Deal-move for a game in-progress #2.
Given deck: `5Sd 4Sd 8Sd 5Sd 3Sd 1Sd 7Sd 4Sd 6Sd 8Sd 2Sd 4Sd 7Sd 7Sd 2Sd 6Sd 5Sd 5Sd 5Sd 6Sd 1Sd 9Sd 8Sd 9Sd 4Sd 7Sd 9Sd 6Sd 9Sd 7Sd 4Sd 6Sd 1Sd 0Sd 8Sd 3Sd`
Given board:

```
#0   #1   #2   #3   #4   #5   #6   #7   #8
--   --   --   --   --   --   --   --   --
0u   0d   2d   7d   4u   5d   3u   6d   1d
     2d   1d   0d        1u        7d   9d
     3d   0d   5u                  9d   0d
     2d   8d                       6d   4u
     8u   2u                       3u
```

Expected deck: `8Sd 2Sd 4Sd 7Sd 7Sd 2Sd 6Sd 5Sd 5Sd 5Sd 6Sd 1Sd 9Sd 8Sd 9Sd 4Sd 7Sd 9Sd 6Sd 9Sd 7Sd 4Sd 6Sd 1Sd 0Sd 8Sd 3Sd`
Expected board:

```
#0   #1   #2   #3   #4   #5   #6   #7   #8
--   --   --   --   --   --   --   --   --
0u   0d   2d   7d   4u   5d   3u   6d   1d
5u   2d   1d   0d   3u   1u   7u   7d   9d
     3d   0d   5u        1u        9d   0d
     2d   8d   5u                  6d   4u
     8u   2u                       3u   6u
     4u   8u                       4u
```

Part 9: Move One or More Cards from One Column to Another

```
int move_card(CardList* board[], CardList* deck, int move)
```

The `move_card` function attempts to perform the game move encoded in the `move` argument using the provided game board and deck of cards. If the attempted move is valid, the function executes it. Valid moves include:

- a deal-move
- moving all of the cards in the donor column to an empty recipient column
- moving all of the cards in the donor column to a nonempty recipient column
- moving some of the cards in the donor column to an empty recipient column. In this case, the new top card in the donor column must be flipped over

- moving some of the cards in the donor column to a nonempty recipient column. In this case, the new top card in the donor column must be flipped over

After executing a deal-move, call `clear_full_straight` *on every column* to clear any full straights formed across the board (this is unlikely, but possible!). After executing a normal move, call `clear_full_straight` on the recipient column to clear any full straight formed. Also, for a normal move, don't forget to adjust the sizes of the donor and recipient columns.

The function takes the following arguments, in this order:

- `board`: an array of 9 pointers to initialized `CardList` structs
- `deck`: a pointer to a valid `CardList`
- `move`: an integer that encodes a game move as described in Part 6

Returns in $v0:

- -1 if a call to `check_move` indicates that the move is illegal in some way
- 1 if the given move was successfully executed

Additional requirements:

- The function must not make any changes to main memory except as necessary.
- The function must call `check_move`, `deal_move`, and `clear_full_straight`.

Example #1: Move one card successfully
```
deck = 9Sd 0Sd 2Sd 5Sd 4Sd 9Sd 0Sd 2Sd 1Sd 0Sd 3Sd 6Sd 9Sd 8Sd 9Sd 5Sd 0Sd
6Sd 5Sd 2Sd 3Sd 3Sd 4Sd 3Sd 2Sd 8Sd 3Sd 7Sd 1Sd 1Sd 7Sd 9Sd 1Sd 9Sd 0Sd
0Sd
```
`move = 0x00070402` (Move cards at and above [2, 4] to column 7)

Given `board`:
```
#0   #1   #2   #3   #4   #5   #6   #7   #8
--   --   --   --   --   --   --   --   --
8d   7d   8d   4d   1d   6d   6d   7d   3d
4d   1d   4d   6d   3d   7d   9d   6d   3d
4d   8d   2d   5d   8d   7d   2d   2d   8d
1d   0d   4d   2d   5d   1d   6d   7d   9u
5u   5u   6u   4u   0u   5u   8u   7u
```
Return value in $v0: 1

Expected `deck`:
```
9Sd 0Sd 2Sd 5Sd 4Sd 9Sd 0Sd 2Sd 1Sd 0Sd 3Sd 6Sd 9Sd 8Sd 9Sd
5Sd 0Sd 6Sd 5Sd 2Sd 3Sd 3Sd 4Sd 3Sd 2Sd 8Sd 3Sd 7Sd 1Sd 1Sd 7Sd 9Sd 1Sd
9Sd 0Sd 0Sd
```
Expected `board`:
```
#0   #1   #2   #3   #4   #5   #6   #7   #8
--   --   --   --   --   --   --   --   --
8d   7d   8d   4d   1d   6d   6d   7d   3d
4d   1d   4d   6d   3d   7d   9d   6d   3d
4d   8d   2d   5d   8d   7d   2d   2d   8d
1d   0d   4u   2d   5d   1d   6d   7d   9u
```

```
5u  5u      4u  0u  5u  8u  7u
                            6u
```

Example #2: Invalid move

`deck = 0Sd 7Sd 4Sd 1Sd 2Sd 1Sd 1Sd 9Sd 5Sd 9Sd 0Sd 8Sd 1Sd 6Sd 4Sd 8Sd 5Sd 8Sd 3Sd 4Sd 4Sd 6Sd 9Sd 5Sd 6Sd 5Sd 9Sd 0Sd 2Sd 7Sd 7Sd 0Sd 3Sd 3Sd 9Sd 5Sd`

`move = 0x00000207` (Move cards at and above [7, 2] to column 0)

Given `board`:

```
#0  #1  #2  #3  #4  #5  #6  #7  #8
--  --  --  --  --  --  --  --  --
9d  9d  7d  2d  0d  1d  2d  6d  0d
6d  8d  4d  7d  6d  8d  0d  4d  8d
3d  1d  7d  2d  3d  1d  3d  3d  5d
5d  2d  4d  1d  5d  6d  7d  7d  9u
3u  8u  4u  0u  8u  6u  2u  2u
```

Return value in $v0: -1

Expected `deck`: `0Sd 7Sd 4Sd 1Sd 2Sd 1Sd 1Sd 9Sd 5Sd 9Sd 0Sd 8Sd 1Sd 6Sd 4Sd 8Sd 5Sd 8Sd 3Sd 4Sd 4Sd 6Sd 9Sd 5Sd 6Sd 5Sd 9Sd 0Sd 2Sd 7Sd 7Sd 0Sd 3Sd 3Sd 9Sd 5Sd`

Expected `board`:

```
#0  #1  #2  #3  #4  #5  #6  #7  #8
--  --  --  --  --  --  --  --  --
9d  9d  7d  2d  0d  1d  2d  6d  0d
6d  8d  4d  7d  6d  8d  0d  4d  8d
3d  1d  7d  2d  3d  1d  3d  3d  5d
5d  2d  4d  1d  5d  6d  7d  7d  9u
3u  8u  4u  0u  8u  6u  2u  2u
```

Example #3: Moving some of the cards in the donor column to an empty recipient column

`deck = 0Sd 7Sd 4Sd 1Sd 2Sd 1Sd 1Sd 9Sd 5Sd 9Sd 0Sd 8Sd 1Sd 6Sd 4Sd 8Sd 5Sd 8Sd 3Sd 4Sd 4Sd 6Sd 9Sd 5Sd 6Sd 5Sd 9Sd 0Sd 2Sd 7Sd 7Sd 0Sd 3Sd 3Sd 9Sd 5Sd`

`move = 0x00060207` (Move cards at and above [7, 2] to column 6)

Given `board`:

```
#0  #1  #2  #3  #4  #5  #6  #7  #8
--  --  --  --  --  --  --  --  --
9d  9d  8u  2d  0u          6d  0d
6d  8d  7u  7u              4d  8u
3u  1d  6u  6u              3u  7u
    2d  5u  5u              2u
    8u  4u  4u              1u
    7u  3u  3u              0u
        2u  2u
        1u
```

Return value in $v0: 1
Expected `deck`: `0Sd 7Sd 4Sd 1Sd 2Sd 1Sd 1Sd 9Sd 5Sd 9Sd 0Sd 8Sd 1Sd 6Sd 4Sd 8Sd 5Sd 8Sd 3Sd 4Sd 4Sd 6Sd 9Sd 5Sd 6Sd 5Sd 9Sd 0Sd 2Sd 7Sd 7Sd 0Sd 3Sd 3Sd 9Sd 5Sd`
Expected `board`:
```
#0   #1   #2   #3   #4   #5   #6   #7   #8
--   --   --   --   --   --   --   --   --
9d   9d   8u   2d   0u        3u   6d   0d
6d   8d   7u   7u             2u   4u   8u
3u   1d   6u   6u             1u        7u
     2d   5u   5u             0u
     8u   4u   4u
     7u   3u   3u
          2u   2u
          1u
```

Example #4: Moving some of the cards in the donor column to a nonempty recipient column
`deck = 0Sd 7Sd 4Sd 1Sd 2Sd 1Sd 1Sd 9Sd 5Sd 9Sd 0Sd 8Sd 1Sd 6Sd 4Sd 8Sd 5Sd 8Sd 3Sd 4Sd 4Sd 6Sd 9Sd 5Sd 6Sd 5Sd 9Sd 0Sd 2Sd 7Sd 7Sd 0Sd 3Sd 3Sd 9Sd 5Sd`
`move = 0x00010102` (Move cards at and above [2, 1] to column 1)
Given `board`:
```
#0   #1   #2   #3   #4   #5   #6   #7   #8
--   --   --   --   --   --   --   --   --
9d   9d   8u   2u        7u   8u   6d   0d
6d   8u   7u             6u   7u   4u   8u
3u        6u             5u        3u   7u
2u        5u             4u        2u
1u        4u             3u        1u
          3u             2u        0u
          2u
          1u
          0u
```
Return value in $v0: 1
Expected `deck`: `0Sd 7Sd 4Sd 1Sd 2Sd 1Sd 1Sd 9Sd 5Sd 9Sd 0Sd 8Sd 1Sd 6Sd 4Sd 8Sd 5Sd 8Sd 3Sd 4Sd 4Sd 6Sd 9Sd 5Sd 6Sd 5Sd 9Sd 0Sd 2Sd 7Sd 7Sd 0Sd 3Sd 3Sd 9Sd 5Sd`
Expected `board`:
```
#0   #1   #2   #3   #4   #5   #6   #7   #8
--   --   --   --   --   --   --   --   --
9d   9d   8u   2u        7u   8u   6d   0d
6d   8u                  6u   7u   4u   8u
3u   7u                  5u        3u   7u
2u   6u                  4u        2u
1u   5u                  3u        1u
```

```
         4u                     2u         0u
         3u
         2u
         1u
         0u
```

Example #5: Moving all of the cards in the donor column to an empty recipient column

`deck = 0Sd 7Sd 4Sd 1Sd 2Sd 1Sd 1Sd 9Sd 5Sd 9Sd 0Sd 8Sd 1Sd 6Sd 4Sd 8Sd 5Sd 8Sd 3Sd 4Sd 4Sd 6Sd 9Sd 5Sd 6Sd 5Sd 9Sd 0Sd 2Sd 7Sd 7Sd 0Sd 3Sd 3Sd 9Sd 5Sd`

`move = 0x00040005` (Move cards at and above [5, 0] to column 4)

Given `board`:

```
#0   #1   #2   #3   #4   #5   #6   #7   #8
--   --   --   --   --   --   --   --   --
9d   9d   8u   2u        7u   8u   6d   0d
6d   8u   7u             6u   7u   4u   8u
3u        6u             5u        3u   7u
2u        5u             4u        2u
1u        4u             3u        1u
          3u             2u        0u
          2u
          1u
          0u
```

Return value in $v0: 1

Expected `deck`: `0Sd 7Sd 4Sd 1Sd 2Sd 1Sd 1Sd 9Sd 5Sd 9Sd 0Sd 8Sd 1Sd 6Sd 4Sd 8Sd 5Sd 8Sd 3Sd 4Sd 4Sd 6Sd 9Sd 5Sd 6Sd 5Sd 9Sd 0Sd 2Sd 7Sd 7Sd 0Sd 3Sd 3Sd 9Sd 5Sd`

Expected `board`:

```
#0   #1   #2   #3   #4   #5   #6   #7   #8
--   --   --   --   --   --   --   --   --
9d   9d   8u   2u   7u        8u   6d   0d
6d   8u   7u        6u        7u   4u   8u
3u        6u        5u             3u   7u
2u        5u        4u             2u
1u        4u        3u             1u
          3u        2u             0u
          2u
          1u
          0u
```

Example #6: Moving all of the cards in the donor column to a nonempty recipient column

`deck = 0Sd 7Sd 4Sd 1Sd 2Sd 1Sd 1Sd 9Sd 5Sd 9Sd 0Sd 8Sd 1Sd 6Sd 4Sd 8Sd 5Sd 8Sd 3Sd 4Sd 4Sd 6Sd 9Sd 5Sd 6Sd 5Sd 9Sd 0Sd 2Sd 7Sd 7Sd 0Sd 3Sd 3Sd 9Sd 5Sd` `move = 0x00060005` (Move cards at and above [5, 0] to column 6)

Given `board`:

```
#0   #1   #2   #3   #4   #5   #6   #7   #8
--   --   --   --   --   --   --   --   --
9d   9d   8u   2u   7u   7u   8u   6d   0d
6d   8u   7u             6u        4u   8u
3u        6u             5u        3u   7u
2u        5u             4u        2u
1u        4u             3u        1u
          3u             2u        0u
          2u
          1u
          0u
```

Return value in $v0: 1

Expected `deck`: `0Sd 7Sd 4Sd 1Sd 2Sd 1Sd 1Sd 9Sd 5Sd 9Sd 0Sd 8Sd 1Sd 6Sd 4Sd 8Sd 5Sd 8Sd 3Sd 4Sd 4Sd 6Sd 9Sd 5Sd 6Sd 5Sd 9Sd 0Sd 2Sd 7Sd 7Sd 0Sd 3Sd 3Sd 9Sd 5Sd`

Expected `board`:

```
#0   #1   #2   #3   #4   #5   #6   #7   #8
--   --   --   --   --   --   --   --   --
9d   9d   8u   2u   7u        8u   6d   0d
6d   8u   7u                  7u   4u   8u
3u        6u                  6u   3u   7u
2u        5u                  5u   2u
1u        4u                  4u   1u
          3u                  3u   0u
          2u                  2u
          1u
          0u
```

Example #7: Deal-move

`deck = 0Sd 7Sd 4Sd 1Sd 2Sd 1Sd 1Sd 9Sd 5Sd 9Sd 0Sd 8Sd 1Sd 6Sd 4Sd 8Sd 5Sd 8Sd 3Sd 4Sd 4Sd 6Sd 9Sd 5Sd 6Sd 5Sd 9Sd 0Sd 2Sd 7Sd 7Sd 0Sd 3Sd 3Sd 9Sd 5Sd`

`move = 0x01000000` (Deal-move)

Given `board`:

```
#0   #1   #2   #3   #4   #5   #6   #7   #8
--   --   --   --   --   --   --   --   --
9d   9d   8u   2u   7u   7u   8u   6d   0d
6d   8u   7u             6u        4u   8u
3u        6u             5u        3u   7u
2u        5u             4u        2u
1u        4u             3u        1u
          3u             2u        0u
          2u
```

```
            1u
            0u
```

Return value in $v0: 1

Expected `deck`: `9Sd 0Sd 8Sd 1Sd 6Sd 4Sd 8Sd 5Sd 8Sd 3Sd 4Sd 4Sd 6Sd 9Sd 5Sd 6Sd 5Sd 9Sd 0Sd 2Sd 7Sd 7Sd 0Sd 3Sd 3Sd 9Sd 5Sd`

Expected `board`:

```
#0    #1    #2    #3    #4    #5    #6    #7    #8
--    --    --    --    --    --    --    --    --
9d    9d    8u    2u    7u    7u    8u    6d    0d
6d    8u    7u    1u    2u    6u    1u    4u    8u
3u    7u    6u                5u          3u    7u
2u          5u                4u          2u    5u
1u          4u                3u          1u
0u          3u                2u          0u
            2u                1u          9u
            1u
            0u
            4u
```

Part 10: Load a Game Board

```
int, int load_game(string filename, CardList* board[], CardList* deck,
                   int[] moves)
```

The `load_game` function opens the file given by the path name filename and uses its contents to initialize `board`, `deck` and `moves`. The argument `board` contains pointers to 9 uninitialized `CardList` structs that will eventually store the contents of the board; `deck` contains a pointer to an uninitialized `CardList`; and `moves` contains a pointer to a buffer of allocated memory guaranteed to be large enough to store all the moves (encoded as integers) from the file.

The file has the format given below. You are guaranteed that, if the file can be successfully opened, its contents are valid:

```
[starting deck of cards]
[array of moves encoded as 4-character strings separated by spaces]
[starting configuration of board]
```

Each line will end with only \n, never with \r\n. The last line of the file will also contain a \n at the end.

As an example, below are the contents of game01.txt, with the deck in blue, array of moves in green, and the board's contents in purple. The deck and list of moves will each be contained on a *single line*. However, each row of the board lies on a separate line. Row #0 of the game board is given on the line after the list of moves, followed by row #1 on the line after that, and so on. As a result, the board is contained in the number of lines equal to the longest column.

```
9d8d7d6d5d4d3d2d1d
7080 0001 0170 0001 1100 2100 3100 4100 5100 6100 7100 8200 8100 7080 6080
1100 0001 5080 4080 3080 2080 1080 0080 0110 0080 1080
1d2u3u4u5u6u7u8u9u
0u              0u
```

Exactly two spaces in the board section of the file indicates an empty spot (so that the board lines up nicely). For example, we see on the last line of the file that only columns 0 and 8 contain cards in row #1.

The layout of the function would look like this:

1. Assuming the file exists, the function calls `init_list` to initialize the deck. Then it can read the first line of the file to initialize the contents of the deck in memory by creating and appending `CardNode` structs to `deck` as needed.
2. Next, the function reads in the second line of the file, which contains the list of moves. For each four-character substring it reads, the function encodes the move as a single integer and writes that integer into the `moves` array. It is the responsibility of the function to keep track of how many moves it reads. Moves may be invalid, but these should still be encoded as integers and included in the `moves` array.
3. Then, the function calls `init_list` on each of the 9 entries in the `board` array. It next reads each line of the file, treating each adjacent pair of characters as encoding a card. Assume that all cards are Spades (♠). Then, the function creates a `CardNode` struct for each card it reads and appends the card to the correct `CardList` in `board`.

The function takes the following arguments, in this order:

- `filename`: the name of the game file to load
- `board`: an array of 9 pointers to uninitialized `CardList` structs (`size` and `head` in all lists are 0)
- `deck`: an uninitialized `CardList`
- `moves`: an uninitialized buffer of memory large enough to store the integer encodings of the moves

Returns in $v0:

- -1 if the file was not found
- 1 if the file was opened successfully

Returns in $v1:

- -1 if the file was not found
- the number of moves encoded in the input file

Additional requirements:

- The function must not make any changes to main memory except as necessary.
- The function must call `init_list` and `append_card`.

Example #1: Load game01.txt

`filename = "game01.txt"`

Expected `deck`: `9Sd 8Sd 7Sd 6Sd 5Sd 4Sd 3Sd 2Sd 1Sd`

Expected `board`:

```
#0   #1   #2   #3   #4   #5   #6   #7   #8
--   --   --   --   --   --   --   --   --
1d   2u   3u   4u   5u   6u   7u   8u   9u
0u                                      0u
```

Expected `moves` (shown here as integers):

```
0x00080007 0x01000000 0x00070100 0x01000000 0x00000101 0x00000102
0x00000103 0x00000104 0x00000105 0x00000106 0x00000107 0x00000208
0x00000108 0x00080007 0x00080006 0x00000101 0x01000000 0x00080005
0x00080004 0x00080003 0x00080002 0x00080001 0x00080000 0x00010100
0x00080000 0x00080001
```

Example #2: Load game02.txt

`filename = "game02.txt"`

Expected `deck`: `0Sd 7Sd 4Sd 1Sd 2Sd 1Sd 1Sd 9Sd 5Sd 9Sd 0Sd 8Sd 1Sd 6Sd 4Sd 8Sd 5Sd 8Sd 3Sd 4Sd 4Sd 6Sd 9Sd 5Sd 6Sd 5Sd 9Sd 0Sd 2Sd 7Sd 7Sd 0Sd 3Sd 3Sd 9Sd 5Sd`

Expected `board`:

```
#0   #1   #2   #3   #4   #5   #6   #7   #8
--   --   --   --   --   --   --   --   --
9d   9d   7d   2d   0d   1d   2d   6d   0d
6d   8d   4d   7d   6d   8d   0d   4d   8d
3d   1d   7d   2d   3d   1d   3d   3d   5d
5d   2d   4d   1d   5d   6d   7d   7d   9u
3u   8u   4u   0u   8u   6u   2u   2u
```

Expected `moves` (shown here as integers):

```
0x00000407 0x00020400 0x00080404 0x00040402 0x00080307 0x00080405
0x00080304 0x00040406 0x00060305 0x00080205 0x00080403 0x00050306
0x00050208 0x00050302 0x00040303 0x00060304 0x00050206 0x00010202
0x00040203 0x00020204 0x00000102 0x00040300 0x00030104 0x00080002
0x00020105 0x00050106 0x00060005 0x00070006 0x00060207 0x00070006
0x00050103 0x00060401 0x00000301 0x00000201 0x00020004 0x00010102
0x00040101 0x00010004 0x00010200 0x00040100 0x00000108 0x00000105
0x00000201 0x00000008 0x00000101 0x00080106 0x01000000 0x00040106
0x00070006 0x00060607 0x00080107 0x00070108 0x00080007 0x00060008
0x00060105 0x00070102 0x00080101 0x01000000 0x00000102 0x00030101
0x00040107 0x00040007 0x00070105 0x00070001 0x00050304 0x00060005
0x00060107 0x00060104 0x00060303 0x00000206 0x00010203 0x00050106
0x01000000 0x00060105 0x00080005 0x00050208 0x00040108 0x00040008
0x00080104 0x00080106 0x00080102 0x00070101 0x00080200 0x00010100
0x00000001 0x00080000 0x00080207 0x00000107 0x00010203 0x01000000
0x00050101 0x00010108 0x00010008 0x00010205 0x00020001 0x00070002
```

```
0x00020203 0x00070103 0x00070100 0x00050002 0x00070106 0x00060104
0x00050004 0x00050000 0x00050007 0x00050003 0x00050006
```

Part 11: Simulate a Game of Spider Solitaire

```
int, int simulate_game(string filename, CardList* board[], CardList* deck,
                       int[] moves)
```

The `simulate_game` function simulates the game of Spider Solitaire as specified in the given input file. As with the arguments to `load_game`, `board` contains pointers to 9 uninitialized `CardList` structs that will eventually store the contents of the board; `deck` contains a pointer to an uninitialized `CardList`; and `moves` contains a pointer to a buffer of allocated memory guaranteed large enough to store all the moves (encoded as integers) from the file.

First the function attempts to load the game stored in the given file by calling `load_game`. If `load_game` fails to find the file, `simulate_game` returns -1, -1 immediately and makes no changes to memory. Assuming the file contents are loaded successfully, and after the `board`, `deck` and `moves` data structures are initialized, `simulate_game` executes moves sequentially, starting at `moves[0]`, until all of the moves are executed or the game is won. Some of the moves encoded in the file could be invalid, which cannot be detected until the function attempts to execute the moves. Any invalid moves in the `moves` array are simply ignored. Then, in $v0, the function returns the number of moves executed. In $v1, the function returns -2 if the game is not won, or 1 if the game is won.

The function takes the following arguments, in this order:

- `filename`: the name of the game file to load
- `board`: an array of 9 pointers to uninitialized `CardList` structs (`size` and `head` in all lists are 0)
- `deck`: an uninitialized `CardList`
- `moves`: an uninitialized buffer of memory large enough to store the integer encodings of all of the moves

Returns in $v0:

- -1 if the file could not be opened
- the number of valid moves that were executed

Returns in $v1:

- -1 if the file could not be opened
- -2 if all moves from the moves array were executed, but the game was not won
- 1 if the game was won (i.e., the game board is cleared and the deck is empty)

Additional requirements:

- The function must not make any changes to main memory.
- The function must call `load_game` and `move_card`.

Example #1: Simulate game01.txt - results in a win
filename = "game01.txt"
File contents:
```
9d8d7d6d5d4d3d2d1d
7080 0001 0170 0001 1100 2100 3100 4100 5100 6100 7100 8200 8100 7080 6080
1100 0001 5080 4080 3080 2080 1080 0080 0110 0080 1080
1d2u3u4u5u6u7u8u9u
0u              0u
```
Expected `deck` at end of simulation: empty
Expected `board` at end of simulation:
```
#0   #1   #2   #3   #4   #5   #6   #7   #8
--   --   --   --   --   --   --   --   --
```
(empty board)
Expected `moves` at end of simulation:
```
7080 0001 0170 0001 1100 2100 3100 4100 5100 6100 7100 8200 8100 7080 6080
1100 0001 5080 4080 3080 2080 1080 0080 0110 0080 1080
```
Return values: 20, 1

Example #2: Simulate game02.txt - results in a win
filename = "game02.txt"
File contents:
```
0d7d4d1d2d1d1d9d5d9d0d8d1d6d4d8d5d8d3d4d4d6d9d5d6d5d9d0d2d7d7d0d3d3d9d5d
7400 0420 4480 2440 7380 5480 4380 6440 5360 5280 3480 6350 8250 2350 3340
4360 6250 2210 3240 4220 2100 0340 4130 2080 5120 6150 5060 6070 7260 6070
3150 1460 1300 1200 4020 2110 1140 4010 0210 0140 8100 5100 1200 8000 1100
6180 0001 6140 6070 7660 7180 8170 7080 8060 5160 2170 1180 0001 2100 1130
7140 7040 5170 1070 4350 5060 7160 4160 3360 6200 3210 6150 0001 5160 5080
8250 8140 8040 4180 6180 2180 1170 0280 0110 1000 0080 7280 7100 3210 0001
1150 8110 8010 5210 1020 2070 3220 3170 0170 2050 6170 4160 4050 0050 7050
3050 6050
9d9d7d2d0d1d2d6d0d
6d8d4d7d6d8d0d4d8d
3d1d7d2d3d1d3d3d5d
5d2d4d1d5d6d7d7d9u
3u8u4u0u8u6u2u2u
```
Expected `deck` at end of simulation: empty
Expected `board` at end of simulation:
```
#0   #1   #2   #3   #4   #5   #6   #7   #8
--   --   --   --   --   --   --   --   --
```
(empty board)
Expected `moves` at end of simulation:
```
7400 0420 4480 2440 7380 5480 4380 6440 5360 5280 3480 6350 8250 2350 3340
4360 6250 2210 3240 4220 2100 0340 4130 2080 5120 6150 5060 6070 7260 6070
3150 1460 1300 1200 4020 2110 1140 4010 0210 0140 8100 5100 1200 8000 1100
6180 0001 6140 6070 7660 7180 8170 7080 8060 5160 2170 1180 0001 2100 1130
```

```
7140 7040 5170 1070 4350 5060 7160 4160 3360 6200 3210 6150 0001 5160 5080
8250 8140 8040 4180 6180 2180 1170 0280 0110 1000 0080 7280 7100 3210 0001
1150 8110 8010 5210 1020 2070 3220 3170 0170 2050 6170 4160 4050 0050 7050
3050 6050
```
Return values: 107, 1

Example #3: Simulate game03.txt - results in a win (some invalid moves in the moves[] array)
filename = "game03.txt"
File contents:
```
9d8d7d6d5d4d3d2d1d
0010 0020 7080 0001 0170 0080 0001 1100 2100 3100 4100 0002 5100 6100 7100
1234 8200 8100 7080 6080 1100 0001 9999 5080 4080 3080 2080 1080 0080 6969
0110 0080 1080
1d2u3u4u5u6u7u8u9u
0u             0u
```
Expected `deck` at end of simulation: empty
Expected `board` at end of simulation:
```
#0   #1   #2   #3   #4   #5   #6   #7   #8
--   --   --   --   --   --   --   --   --
```
(empty board)
Expected `moves` at end of simulation:
```
0010 0020 7080 0001 0170 0080 0001 1100 2100 3100 4100 0002 5100 6100 7100
1234 8200 8100 7080 6080 1100 0001 9999 5080 4080 3080 2080 1080 0080 6969
0110 0080 1080
```
Return values: 20, 1

Example #4: Simulate game04.txt - results in a loss
filename = "game04.txt"
File contents:
```
6d0d5d4d7d4d6d2d2d4d8d2d9d7d0d9d1d2d1d5d8d4d6d7d5d6d7d8d5d9d8d3d7d3d3d1d
2280 1908 2460 6430 3450 5470 7400 0480 8310 3440 4480 0380 8350 5410 2310
1430 3410 1430 3410 1430 3410 1430 3410 1430 3410 1430 3410 1430 3410 1430
3410 1430 3410 1430 3410 1430 3410 1430 3410 1430 3410 1430 3410 1430 3410
1430 3410 1430 3410 1430 3410 5555 1111 1223 1231 2412 3214 1211 1430 3410
1430 3410 1430 3410 1430 3410 1430 3410 1430 3410 1430 3410 1430 3410 1430
3410 1430 3410 1430 3410 1430 3410 1430 3410 1430 3410 1430 3410 1430 3410
1430 3410 1430 3410 1430 3410 1430 3410 1430 3410 1430 3410 1430 3410 1430
3410 1430 3410 1430 3410 1430 3410
6d9d2d0d3d1d7d1d3d
8d1d3d9d2d4d1d5d6d
8d9d0d0d9d0d2d3d6d
0d5d4d5d4d0d7d8d7u
6u9u1u3u4u8u2u5u
```

Expected `deck` at end of simulation: `6Sd 0Sd 5Sd 4Sd 7Sd 4Sd 6Sd 2Sd 2Sd 4Sd 8Sd 2Sd 9Sd 7Sd 0Sd 9Sd 1Sd 2Sd 1Sd 5Sd 8Sd 4Sd 6Sd 7Sd 5Sd 6Sd 7Sd 8Sd 5Sd 9Sd 8Sd 3Sd 7Sd 3Sd 3Sd 1Sd`

Expected `board` at end of simulation:

```
#0   #1   #2   #3   #4   #5   #6   #7   #8
--   --   --   --   --   --   --   --   --
6d   9d   2d   0d   3d   1d   7d   1d   3d
8d   1d   3d   9d   2d   4d   1d   5d   6d
8u   9d   0u   0d   9d   0d   2d   3d   6u
     5u        5u   4u   0u   7u   8u
     4u
```

Expected `moves` at end of simulation:

```
2280 1908 2460 6430 3450 5470 7400 0480 8310 3440 4480 0380 8350 5410 2310
1430 3410 1430 3410 1430 3410 1430 3410 1430 3410 1430 3410 1430 3410 1430
3410 1430 3410 1430 3410 1430 3410 1430 3410 1430 3410 1430 3410 1430 3410
1430 3410 1430 3410 1430 3410 5555 1111 1223 1231 2412 3214 1211 1430 3410
1430 3410 1430 3410 1430 3410 1430 3410 1430 3410 1430 3410 1430 3410 1430
3410 1430 3410 1430 3410 1430 3410 1430 3410 1430 3410 1430 3410 1430 3410
1430 3410 1430 3410 1430 3410 1430 3410 1430 3410 1430 3410 1430 3410 1430
3410 1430 3410 1430 3410 1430 3410
```

Return values: 100, -2

Academic Honesty Policy

Academic honesty is taken very seriously in this course. By submitting your work for grading you indicate your understanding of, and agreement with, the following Academic Honesty Statement:

1. I understand that representing another person's work as my own is academically dishonest.
2. I understand that copying, even with modifications, a solution from another source (such as the web or another person) as a part of my answer constitutes plagiarism.
3. I understand that sharing parts of my homework solutions (text write-up, schematics, code, electronic or hard-copy) is academic dishonesty and helps others plagiarize my work.
4. I understand that protecting my work from possible plagiarism is my responsibility. I understand the importance of saving my work such that it is visible only to me.
5. I understand that passing information that is relevant to a homework/exam to others in the course (either lecture or even in the future!) for their private use constitutes academic dishonesty. I will only discuss material that I am willing to openly post on the discussion board.
6. I understand that academic dishonesty is treated very seriously in this course. I understand that the instructor will report any incident of academic dishonesty to the University's Academic Judiciary.
7. I understand that the penalty for academic dishonesty might not be immediately administered. For instance, cheating on a homework assignment may be discovered and penalized after the grades for that homework have been recorded.

8. I understand that buying or paying another entity for any code, partial or in its entirety, and submitting it as my own work is considered academic dishonesty.
9. I understand that there are no extenuating circumstances for academic dishonesty.

How to Submit Your Work for Grading

To submit your hwk5.asm file for grading:

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

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

No worries! Just follow the steps again. We will grade only your last submission.