Advanced C Programming

Fall 2017 :: ECE 264 :: Purdue University

This is for Fall 2017 (2 years ago) only.

Card shuffling #2 (A)

Due 10/13

Performing k rounds of shuffling

In HW07, your program gets a deck of cards and shuffle this deck once. The program prints the possible orders (with repetitions) after one round of shuffling. You may have noticed that shuffling only once does not produce all possible orderings of a deck of cards. For example, if the original deck has 4 cards, the number of all possible orderings is 24. However, shuffling a deck of cards once will give us only 14 orderings, not 24. Moreover, some of the 14 results are the same. Note that 24 is 4!, and 14 is 2^4 - 2. If we are dealing with 5 cards, there are 5! possible orderings, but shuffling once will give us only (2^5 - 2) orderings (with repetitions).

This assignment extends the exercise to print the possible orderings of a deck after k rounds of shuffling, using the function

```
void repeat_shuffle (CardDeck orig_deck, int k);
```

where the first parameter is the original deck (orig_deck) and the second parameter is the number of rounds of shuffling to be performed. We assume that a round of shuffling is always performed on a complete deck of cards. It involves dividing a deck of cards into non-empty upper and lower decks, and interleaving the two decks into a complete deck again. The next round of shuffling, if any, should be performed on the complete (shuffled) deck, and the process continues until all k rounds of shuffling have been performed.

The sample output for a deck A 2 3 4 shuffled 3 times is given as a reference. Shuffling once produces 14 (= 2^4 - 2) lines. For each of these orderings, we perform another shuffling to produce 14 possible orderings (with repetitions). Therefore, the second round of shuffling will produce 14*14 possible orderings (with repetitions). Consequently, shuffling 3 times produces 2744 (14*14*14) lines altogether. It is acceptable that the possible orderings produced by your program do not match the order in which the possible orderings appear in the listing produced by the instructor's code. However, when your listing and the instructor's listing are sorted, they should match line by line. Please print only the final results (i.e., all possible orderings with repetitions after all k rounds of shuffling). Do not print the intermediate results.

Tips

It is likely that your *helper* functions used in HW07 require some adjustments. (The `divide(…)`, `interleave(…)` and `shuffle(…)` function signatures should not change.) More specifically, you probably have to add some more parameters to the helper functions written for HW07 in order to

keep track of how many times the cards have been shuffled or how many more rounds of shuffling remain. ~~One purpose of this assignment is to encourage you to think about how to modify the program written for HW07 for this assignment. When you reuse the code from HW07, be sure to remove the `#ifdef-#endif` pairs in the program.~~

Tips for getting started

1. `repeat_shuffle(…)` is similar to shuffle in HW07, except that it has an additional parameter to keep track of the number of rounds (int k) to shuffle cards.
2. `repeat_shuffle(…)` can be written as a part of a recursive paradigm, but it does not call itself. (In all exercises so far, all recursive functions call themselves directly.) Here, you may want to consider the case that `repeat_shuffle(…)` calls another function, which then calls `repeat_shuffle(…)`, but with a smaller problem size. (What is the problem size in this recursive paradigm?) In other words, `repeat_shuffle(…)` is called recursively in an indirect fashion.
3. When the number of rounds of shuffling is zero, what should `repeat_shuffle(…)` do? This is the terminating condition for `repeat_shuffle(…)`. Recall that this function is supposed to print possible outcomes (with repetitions) after k rounds of shuffling. If k = 0, what should be printed? Are there other terminating conditions?
4. If the number of rounds (int k) of shuffling is greater than zero, `repeat_shuffle(…)` should perform what you did in HW07, i.e., find all possible pairs of dividing cards, and for each pair, perform interleaving.
5. When you are done with interleaving an upper deck and lower deck and obtain a complete deck of cards, you have completed one round of shuffling. Now, you have to perform (k-1) more rounds of shuffling. What function can you call to perform (k-1) rounds of shuffling of a deck of cards?

If you know the answers to these questions, your shuffle.c in ~~HW07~~ HW08 should look very similar to shuffle.c in HW07.

You can also have an implementation that uses an array to store the results of one round of shuffling. Another round of shuffling can then be applied to each deck in the array. You would have to allocate space for the storage of intermediate results, and you should also be responsible for freeing the space when you no longer need them.

Tips for testing

First, check whether you have correct number of lines for the output for a given deck size and `k` (# of rounds).

You may wish to modify your `main(…)` to accept two command-line arguments: the number of cards in a deck (`argv[1]`), and the number of rounds of shuffling to be performed (`argv[2]`).

Too many rounds of shuffling will generate a large number of possible orderings. This may affect your testing in three ways:

1. Your implementation may require you to allocate memory to store the orderings. If the number of rounds of shuffling is too high, you may run out of memory. (If you run out of memory, you should free the memory you have allocated so far, and return `EXIT_FAILURE` from the main function). Therefore, it is important that after you call malloc, you always check to see whether the malloc function is successful in returning a non-NULL address.
2. For your testing, you probably want to save the output to a file.
   ```
   ./shuffle 4 3 > ordering_output
   ```
   The screen output will be redirected and be stored in the file ordering_output. Too many rounds of shuffling may result in an extremely large file, and you may run out of disk quota.

Therefore, you should not attempt a large number of rounds of shuffling during testing AND also redirect the screen output to a file. It is fine to attempt a large number of rounds of shuffling, without redirecting the screen output to a file, to stress test your memory allocation, if necessary.

3. Depending on your implementation, you may have a deep recursion that exhaust all space on the call stack.

Requirements

1. Your submission must contain each of the following files, as specified:

file	contents	
shuffle.c	functions	**`repeat_shuffle`**`(CardDeck orig_deck, int k)` → *return type:* void • `orig_deck` contains the number of cards • The number of upper-low deck pairs should be the number of cards - 1. • If (k ≤ 0), no shuffling, print the only possible outcome. • Otherwise, for each pair of upper and lower decks, interleave the cards • In `interleave(…)`, when the newly shuffled deck is complete, you will perform another k-1 rounds of shuffling with the new deck. • Print only the results obtained after k rounds of shuffling
		`divide``(CardDeck orig_deck, CardDeck * upper_decks, CardDeck * lower_decks)` → *return type:* void Divide a deck into into pairs of upper and lower decks. • Follow the spec in HW07.
		`interleave``(CardDeck upper_deck, CardDeck lower_deck)` → *return type:* void Print all possible interleavings of `upper_deck` with `lower_deck`. • Follow the spec in HW07.
		`shuffle``(CardDeck orig_deck)` → *return type:* void Generate all possible decks that could result from *one* shuffle. • Follow the spec in HW07.

2. Submissions must meet the code quality standards and the course policies on homework and academic integrity.

Submit

To submit HW08, type `264submit HW08 shuffle.c` from inside your hw08 directory.

> In general, to submit any assignment for this course, you will use the following command:
>
> `264submit` *ASSIGNMENT* *FILES…*
>
> Submit often and early, even well before you are finished. Doing so creates a backup that you can retrieve in case of a problem (e.g., accidentally deleted your files).
>
> To retrieve your most recent submission, type `264get --restore` *ASSIGNMENT* (e.g., `264get --restore hw08`).
>
> To retrieve an earlier submission, first type `264get --list` *ASSIGNMENT* to view your past submissions and find the timestamp of the one you want to retrieve. Then, type `264get`

`--restore -t` *TIMESTAMP* *ASSIGNMENT*.

Scores will be posted to the Scores page after the deadline for each assignment.

Pre-tester ●

The pre-tester for HW08 has been released and is ready to use.

Using the pretester

The pretester is a tool for checking your work after you believe you are done, and before we have scored it. It is not a substitute for your own checking, but it may help you avoid big surprises by letting you know if your checking was not adequate. To use the pre-tester, first submit your code. Then, type the following command. (*Do this only after you have submitted, and only after you believe your submission is perfect.*)

```
264test hw08
```

Do not ask TAs or instructors which tests you failed.

Keep in mind:

- Pre-testing is intended only for those who believe they are done and believe their submission is perfect.
- The pre-tester is not part of the requirements of this or any other assignment.
- You are responsible for reading the assignment carefully, and ensuring that your code meets all requirements.
- Feedback is limited, to ensure that everyone learns to test their own code.
- If your code is failing some tests, review *your tests* and make sure they are comprehensive enough to catch any bugs (deviations from requirements). Follow the tips given by the pre-tester.
- Code quality issues are not reported by the pre-tester; writing clean code is something you must learn to do from the start, not a clean-up step to do at the end.

Logistics:

- If we discover that we have not checked some significant part of the assignment requirements, we may add additional tests at any time up to the point when scores are released.
- The pre-tester will only be enabled after much of the class has submitted the assignment, and at least a few people have submitted perfect submissions. This is to allow us to test the pre-tester.
- The pre-tester checks your most recent submission. You must submit first.
- You may be limited to running the pre-tester ≤24 times in a 24-hour period. (This is not implemented yet but will be added.)

Q&A

Answers to common questions may be posted here later.

Updates

10/10/2017 Modified starter code to include headers for HW07 functions in shuffle.h. Clarified that only the helper functions from HW07 change. Improved layout of pre-tester and submission instructions. Added table of contents.

10/11/2017 Added note about pretester. Further clarified that parameters may be changed only on *helper* functions.
