From “Faro Shuffle” to “Fhaurfof ISe”

Take a deck of 52 cards. Split the deck into two equal halves of 26 cards each. Re-assemble the two halves so that the cards are perfectly interwoven so that the top card of the starting deck is the top card of the shuffled deck. This is known as the Faro shuffle, a pinnacle skill of any magician or card dealer.

ASIDE: Amazingly or perhaps unsurprisingly, depending on perspective, the Faro shuffle also has algorithmic applications in parallel machines [1], as well as being a derivation from group theory [2]. While the theoretic foundations are lovely, there is no need to understand them in order to see their elegant implications. For example, one of the most interesting features is that, with a deck of 52 cards, 8 Faro shuffles will restore the deck to its original order!

A lesser known derivation, which is the subject of this practical, is the in/out shuffle [3]. Its past applications include efficiently moving data around in memory. The in/out consists of two types of faro shuffles. The difference between them is the order in which cards from the two halves alternate as they are re-assembled:

out-shuffle – the top card of the deck remains as the top card after the shuffle (this is the original faro as described above);

in-shuffle – the top card of the deck occupies the second position in the deck.

Now, imagine the following. Spread the deck of cards and ask a friend to select any card. Gather the cards and ensure that the selected card is the top-most card in the deck. Executed with perfect precision, a magician can use the following technique to ‘guess’ a person’s card.

1. Ask the friend to give a number, k, and tell them only to think of this number.
2. Convert the number to its binary representation.
3. Inspect bits from left to right, performing an *in-shuffle* for every 1-bit, and an *out-shuffle* for every 0-bit.

After all shuffles, the card that was top-most in position-0 will be located in the k^{th} position in the deck! For example, suppose that the top card is the $7\clubsuit$ and $k = 9$:

$$\begin{tikzcd}
1 \arrow[d] & 0 \arrow[d] & 0 \arrow[d] & 1 \arrow[d] \\
in & out & out & in
\end{tikzcd}$$

After an in-shuffle, two out-shuffles, and another in-shuffle, the $7\clubsuit$ will be the 10^{th} card in the deck (i.e. in position 9 when counting from 0).

[1] H. S. Stone, “Parallel Processing with the Perfect Shuffle,” in IEEE Transactions on Computers, 20(2), pp. 153-161, Feb. 1971. doi: 10.1109/T-C.1971.223205

[2] C. Ronse, “A generalization of the perfect shuffle,” Discrete Mathematics, Elsevier, 47, pp. 293-306, 1983. 10.1016/0012-365X(83)90100-0

[3] S.B. Morris and R.E. Hartwig, “The generalized faro shuffle,” Discrete Mathematics, Elsevier, 15(4), pp. 333-346, 1976. doi.org/10.1016/0012-365X(76)90047-9

Requirements & Submission

Write an implementation of the trick described above! For `stacscheck` purposes, the program executable must be named `faro_shuffle` and interact as follows:

0. Accept a command-line argument that is either “RANKSUIT” or “NUMERICAL”. (“NUMERICAL” is for an optional extension; it should be handled, but may otherwise be ignored.)
1. Read two integer values for (i) `size` of the deck and (ii) desired position `k`.
2. Read `k` card-value entries from `stdin`. (see ‘Promises’ section, below).
3. Output the deck contents after *every* shuffle (an output function is provided).
4. Allow new decks and executions by repeating 1-3 until the user types “-1” for the `size`.

The cards’ ranks and suits will be provided by `stacscheck`. An example execution appears below:

```
$ faro_shuffle RANKSUIT
  52
  9
  <52 cards given at command line
  IN: <contents of shuffled deck appear on screen> EoD
  OUT: <contents of shuffled deck appear on screen> EoD
  OUT: <contents of shuffled deck appear on screen> EoD
  IN: <contents of _final_ shuffled deck appears on screen> EoD
  -1
$
```

The final submission should be a zipfile submitted via MMS that contains the following:

- A short report in PDF format that describes design choices and data structures; implementation and testing; as well as any problems encountered and lessons learned.
- Your source code, with a Makefile. This must be in a directory called ‘Practical2-Faro’.

The only source requirements are for,

- a file named `faro_shuffle.c` that contains the program entry point, i.e. main(),
- the executable produced by make to be named `faro_shuffle`.

Any/all additional `.c` and `.h` files are left to your design!

Promises from `stacscheck`

For the purposes of interacting with `stacscheck`:

- The number of input values will always be the `size` given.
- The `size` will always be non-negative, and `size` $mod\ 2 == 0$ will always be true.

- Given the RANKSUIT option, `size` ≤ 52 will always be true.
- RANKSUIT entries will be space-separated, each consisting of two characters, {2, 3, .., 9, 0, J, Q, K} to represent rank and {C, H, S, D} to represent the suit; e.g. `AC` for ace of clubs, `JH` for jack of hearts, `0S` for the 10 of spades[4].

Finally, a `stacscheck`-compliant output function is provided in "print_faro_val.h" alongside a matching object file. Please read the documentation in the header file for usage instructions.

Caveat Emptor:
The input order must be preserved, i.e.`stacscheck` will assume that first input is the top-most card, while the last entry is the bottom-most. This has implications for your implementation.

Constraints

Data structures, functions, and organisation (into header and source files) are left to the designer. However, only the following system libraries may be included (additional info via man-pages or at `http://bit.ly/2EJUVcn`):

- `<stdio.h>`
- `<stdlib.h>`
- `<stdbool.h>`
- `<assert.h>`
- `<string.h>`
- `<math.h>` (for base-10 `log()` function, only)
- `<limits.h>`
- `<time.h>` (time can be used for random seed, if randomness is needed for an extension)

Optional: Additional libraries are permitted only if for the purpose of implementing unlisted extensions.

Hints and Guiding Observations

Keep in mind, or ask yourselves, the following:

Encapsulation can be a friend. Objects and classes are absent from C. Even so, the existing constructs and conventions can be used to encapsulate declarations, data, and functions.

Output to the screen will, at least in part, be determined by the way that data is stored and/or organised (see "CAVEAT EMPTOR", above).

Attack the problem in parts. There are three distinct components that can be solved in isolation: (i) I/O with data storage and management, (ii) use of k to determine shuffles, (iii) the in/out shuffles, themselves.

Re-implementations are possible, likely, and even recommended! For example, it may be easier to start with the familiarity of arrays before attempting alternatives (see 'Classes' below). Practical-0 gave clear evidence and example that the 'back-end' of a program can be implemented without altering the main logic.

[4]Character encodings are very important, especially in the real world; in C, non-ascii is also non-trivial.

(Possible) Extensions

The list of possible extensions includes, though by no means is limited to, the following. *Full functionality of the standard specification is a requirement*; otherwise, extensions will be diminished in value or ignored entirely.

- What about decks of cards larger than 52? The NUMERICAL command-line argument indicates a deck of arbitrary length, up to `UINT_MAX`. Input entries will be space-separated, non-negative in value, and guaranteed to fit in an `unsigned`.

The NUMERICAL option will be used by `stacscheck`. *Both ranksuit and numerical arguments must be handled in the one single implementation.*

Any other non-`stacscheck` extensions should be implemented in a separate sub-directory, clearly indicated and discussed in the report.

- Write a UTF-8 compliant program, one that stores all chars and strings in UTF-8, with corresponding functions. This will allow for suits to be represented as ♣, ♡, ♢, ♠.
- (Minor) Learn to use `valgrind` and, in the report, show the manner in which it was used to find memory leaks (or that there are none).
- The old idiom should be, "Death and taxes *and trade-offs*. What are the trade-offs to the classes of solution (see below)? Find quantitive ways to take measurements that show those trade-offs. Clearly, more than one class or type of solution is necessary.

Sometimes programs can be written to quickly answer questions, or at least point to evidence, where good solutions or answers are missing. One classic and very simple example is the Monte Carlo method for calculating π that, if nothing else, is demonstrative of the power of "flipping a coin."

With respect to the in/out shuffle, there are two open problems where the answers are unknown, at least to me:

- Some fixed number of out-shuffles will always return a deck to its original order; similarly so with in-shuffles (albeit a much greater number). What about a mix of in/out shuffles? Devise and document some appropriate tests. Note that randomness in this context should be avoided.
- Faro shuffles are impossibly difficult to execute in practice. What happens if the shuffle has imperfections? Can the outcomes somehow be characterized? For example, it may be that an imperfect inter-leaving has no effect if it always occurs after the k^{th} position. Devise and document some appropriate tests. Note that, unlike the preceding question, randomness in this context is useful.

Feel free to discuss these or others with tutors. Make sure to describe and detail these or other extensions in your report.

Solution ‘Classes’ and Assessment

Three general classes of implementation exist, listed below in increasing levels of difficulty. Each is followed by its assessment cap (yes, that includes extensions):

- **Static or automatic array** implementations will be best implemented with an array of size 52, for use even when `size < 52`. The array should be re-used. [Capped at 14.]

For the remaining heap/dynamic memory implementations, all heap memory should be freed and re-allocated between executions, i.e. before exit or before accepting a new deck.

- **Heap/dynamic array** implementations that `malloc` once for a whole deck benefit from contiguous memory, and index cards with array-notion ’[]’. [Capped at 15.]
- **Heap/dynamic array** implementations that `malloc` once for a whole deck and use pointers to navigate. While there is no appearance of arrays, the implementation still benefits from contiguous memory. [Capped at 16.]
- **Heap/dynamic memory** implementations avoid array-based structures entirely for the purpose of storing card input[5]

Any storage not used to hold cards and/or decks is exempt from the above.

Work within your class of comfort:
A fully-functional automatic array implementation will always trump a partially-functional anything else; similarly a fully-functional contiguous heap memory (array) implementation will trump a partially-functional dynamic memory implementation.

Assessment & Rubric

Generally speaking, source and reports will be graded primarily along the following criteria:

Scope	The extent to which code implements the features required/specified
Correctness	The extent to which code is consistent with the specs and bug-free.
Design	The extent to which code is well written, ie. clearly, succinctly, elegantly, or logically.
Style	The extent to which code is readable, eg. comments, indentation, apt naming

There is no fixed weighting between the different parts of the practical. So that students might be better prepared for subsequent practicals, general feedback will be offered according to guidelines published in the Student Handbook at,

`https://info.cs.st-andrews.ac.uk/student-handbook/learning-teaching/feedback.html`

Work must be submitted via MMS by the deadline. The standard lateness penalties apply to coursework submitted late, as indicated at

`https://info.cs.st-andrews.ac.uk/student-handbook/learning-teaching/assessment.html#lateness-penalties`

As a reminder, the relevant guidelines on good academic practice are outlined at: `http://www.st-andrews.ac.uk/students/rules/academicpractice/`

[5]Recall from lectures that the exclusion of arrays leaves only one other storage structure.