Although this exercise isn't worth any points, it gives you valuable experience thinking about space/time tradeoffs. You're almost definitely going to have to complete the exercises to succeed in the course.

Problem 1 – Shuffle the Deck

In the Why We Start at Zero lecture in the second course in the specialization, we learned that we can calculate the address of any element in an array by doing some multiplication and addition. That means accessing a particular element is very fast.

Shuffling a deck doesn't take much time, especially for a computer, but it certainly takes more time than calculating a memory address.

How could you use lots of space, but very little time, to quickly get a shuffled deck when you need one? Don't worry about how practical your solution would be; this is a "thought exercise"!

Solution

My (admittedly brute force) solution is to store every possible deck configuration as an element in an array. This array would have 52 * 51 * 50 * ... * 2 * 1 elements (that's 52! different elements for the math inclined).

Of course, this isn't at all practical because 52! is approximately $8.07 * 10^{67}$ array elements, each of which references a deck of 52 cards; that's a LOT of memory!

Problem 2 – Speaking of Multiplication ...

As you might suspect, basic math in computer CPUs (Central Processing Units) has been one of the areas that have been really optimized to minimize CPU cycles as much as possible. Since each CPU cycle is a specific unit of time, that means operations like multiplication are super fast.

Let's say, however, that you don't really believe this, and you've decided to use lots of space to try to make multiplication faster.

How might you do this for multiplying integers between 1 and 100? Would your approach be feasible in practice?

Solution

My solution is to store the multiplication results in a two-dimensional array. That way, we can look up a particular product using the two numbers we're multiplying to determine the row and column of the element that contains the product of those two numbers.

This approach would certainly be feasible (but it wouldn't be faster than just letting the CPU multiply the numbers!). The array would have 100 * 100 = 10,000 elements, each of which is 4 bytes (for an int). That's less than 40K of memory.

Could you use the same approach for multiplying floats between 1 and 100? How feasible would it be?

Solution

We could use the same solution for floats because there are a discrete number of floating point numbers we can store between 1 and 100 in the computer (remember $2^b = n$ from the first course in the specialization). It definitely wouldn't be feasible, though, because it would take massive amounts of space to store all the possible products.