Combinations Continued

In our first entry, we used recursion to generate all possible combinations of pocket twos from a standard, fifty-two card deck. It served as a nice introduction to some core features of the language – type signatures, function definitions, and recursion.

Although it served as a nice introduction, we can simplify the code outlined in combinations\_1. In fact, we can return all combinations k taken from a collection n in a single line, without recursion. For example, consider the list of integers zero to three.

nums = [(x,y) | x <- [0..3], y <- [x..3], x /= y]

Printing nums to standard output will show that we have exhausted all possible combinations of numbers zero through three. We can visualize the comprehension in the following manner:

x = 0; y = 0,1,2,3 => (0,1), (0,2), (0,3)

x = 1; y = 1,2,3 => (1,2), (1,3)

x = 2; y = 2,3 => (2,3)

x = 3; y = 3 => ( )

We can apply this same concept to a list of strings. For example, we can use the subscript operator - !! - in conjunction with the comprehension logic outlined in nums to arrive at all combinations of pocket twos.

twos = ["2c","2d","2h","2s"]

pocketTwos = [(twos !! x, twos !! y) |

x <- [0..length twos - 1],

y <- [x..length twos -1],

x /= y]

The subscript operator returns the value associated with an index. Therefore, passing x and y to the subscript operator will return their concomitant twos.

As outlined on Hoogle, a Haskell API search engine, the subscript operator is a partial function, meaning it is not capable of handling all possible arguments. Alternative functions are often recommended in lieu of !! to avoid runtime errors. Bound errors – requesting an index value that is not present in the list – are a common side effect of index operations across programming languages. Alternatively, we can use one of Haskell’s implementations of a hash map – a collection of key => value pairs – to arrive at the same collection of twos.

deuces = Map.fromList (zip [0..] twos)

pocketTwosSnd = [(fromMaybe "0" (Map.lookup x deuces), fromMaybe "0" (Map.lookup y deuces)) |

x <- [0..length deuces - 1],

y <- [x..length deuces - 1],

x /= y]

First, we create a list of tuples of the type [(Integer, String)]. This is accomplished by passing two lists to the zip function, which pairs elements from two lists until one of the lists is exhausted. In our case, numbers zero through N are paired with the string representations of twos. This list of tuples is then passed to the fromList function to generate our hash map, “deuces.”

Once our hash map has been instantiated, we use our comprehension logic to create a list titled pocketTwosSnd, although the information before the guard is slightly more involved than with the subscript operator.

The Map type returns two values – “just” and “nothing,” otherwise known as “maybe” types. The former indicates that the key exists in the Map, while the latter indicates the opposite. For instance, “Map.lookup 0 deuces” returns “just 2c,” of type “maybe String.”

We can retrieve the underlying value associated with “maybe” by calling the function fromMaybe. fromMaybe retrieves “just” values from the map, returning a default output – in our case “0” – in the event that “nothing” is found.

In summary, pocketTwosSnd composes a tuple by calling the lookup function, once with x and once with y, and passing the results to fromMaybe.

And there you have it. Some alternative solutions for counting ducks.