Overview

The goals of our first Haskell script are straightforward: first, return k items taken from collection n without replacement; second, return k items taken from collection n with replacement; and third, return k items taken from collections (n, m), where collections n and m do not share elements.

For the time being, we are not concerned about performance, nor are we concerned with writing elegant code. Simply put, we need our first foray into functional programming to yield the expected results.

Since an understanding of combinatorics is essential to poker, our examples will revolve around starting hands in No Limit Texas Holdem. Our first and second goals are explored by determining possible combinations of pocket twos, while our third goal is explored by returning possible combinations of Ace-King – poker’s most famous drawing hand.

Pocket Twos: without and with Replacement

Our approach to goals one and two will begin with list comprehensions. We will create a list, “pocketTwos,” with the comprehension:

pocketTwos = [(x,y) | x <- ["2c","2d","2h","2s"], y <- ["2c","2d","2h","2s"], x /= y]

Simply put, this reads, “ return all tuples (x,y), such that x belongs to the set {"2c","2d","2h","2s"}, y belongs to the set {"2c","2d","2h","2s"}, and x does not equal y.”

Any poker player will notice an error with this approach when we print pocketTwos to standard output – the list contains twelve elements. A cursory glance at our tuples shows that we have returned all permutations of pocket twos. That is, we have returned (“2c”, “2d”) in addition to (“2d”, “2c”). We are not interested in this distinction, so we must change course.

We posit that the most obvious approach to our problem is to simply remove duplicate tuples from pocketTwos. In order to remove these duplicates, we must establish equality between differently ordered tuples – (“2c”, “2d”) and (“2d”, “2c”) – while creating a mechanism for removing them from our list. We will accomplish this with two functions: compareTuples and removeTuples.

Type Signatures

Haskell functions generally begin with type signatures, which provide an overview of the types associated with a function’s parameters and return value. Additionally, they describe any constraints placed upon the function’s types. The type signature for our first function, compareTuples, is:

compareTuples :: Eq a => (a,a) -> (a,a) -> Bool

This signature reads, “Type ‘a’ belongs to the Eq class. A pair of two-tuples comprising elements of type ‘a’ is passed to compareTuples, which returns a Boolean value.” The signature’s first component, “Eq a,” is a constraint placed upon the function – essentially, we must be able to determine equality between instances of type “a.”

The type signature of our second function, removeTuples, is:

removeTuples :: Eq a => [(a,a)] -> [(a,a)]

This reads, “Type ‘a’ belongs to the Eq class. A list of two-tuples having type ‘a’ is passed to removeTuples, and a list of two-tuples having type ‘a’ is returned to the caller.”

Function Definitions

Now that we have established our type signatures, we must define our functions. We begin with compareTuples.

As outlined in our type signature, compareTuples takes a pair of two-tuples. We will refer to these tuples as x and y. We must establish equality regardless of order. As it turns out, this can be accomplished compactly:

compareTuples x y | x == y = True

| x == (snd y, fst y) = True

| otherwise = False

The functions reads, “if x is equal to y, then return True; if x is equal to the tuple comprising y’s second element followed by y’s first element, then return True; otherwise, return False.” Compact indeed!

Our definition for removeTuples requires a modicum of nuance. We must determine whether any given element in our list of tuples appears elsewhere in the list. To that end, Haskell supplies the following notation: (x:xs), such that x is the first element in the list, and xs is the remainder of the list.

How do we compare x with any given element in xs? Moreover, how do we continue to filter for duplicates after our first comparison is made? We must use recursion – that is, our function must call itself. Let’s examine the function in full:

removeTuples [ ] = [ ]

removeTuples (x:xs) = x : removeTuples y

where

y = [z | z <- xs, not (compareTuples x z)]

Lines two through four read, “concatenate the list comprising element x with the result of removeTuples y, where y comprises all elements z such that z belongs to xs and z is not equal to x.” Let this sink in for a moment. While this is not intuitive – at least for me – it does make sense when you visualize the definition.

Let’s consider the pairs of Aces taken from the set {“Ac”, “Ad”, “Ah”}. This yields the permutations (“Ac”, “Ad”), (“Ac”, “Ah”), (“Ad”, “Ac”), (“Ad”, “Ah”), (“Ah”, “Ac”), (“Ah”, “Ad”). Our function declares the following:

1. Concatenate [(“Ac”, “Ad”)] with removeTuples [(“Ac”, “Ah”), (“Ad”, “Ah”), (“Ah”, “Ac”), (“Ah”, “Ad”)]. Note that (“Ad”, “Ac”) is absent from the list passed to removeTuples.
2. Concatenate [(“Ac”, “Ah”)] with removeTuples [(“Ad”, “Ah”),(“Ah”, “Ad”)]. Note that (“Ah”, “Ac”) is absent from the list passed to removeTuples.
3. Concatenate [(“Ad”, “Ah”)] with removeTuples [ ]. Once again, a removal has taken place, and we are left with an empty list, which is passed to removeTuples.
4. [ ] ++ [(“Ad”, “Ah”)] = [(“Ad”, “Ah”)]
5. [(“Ad”, “Ah”)] ++ [(“Ac”, “Ah”)] = [(“Ad”, “Ah”), (“Ac”, “Ah”)].
6. [(“Ad”, “Ah”), (“Ac”, “Ah”)] ++ [(“Ac”, “Ad”)] = [(“Ad”, “Ah”), (“Ac”, “Ah”), (“Ac”, “Ad”)].

The nuance in this definition lies in the declaration “removeTuples [ ] = [ ].” That is, passing an empty list into removeTuples will return an empty list. This is called the base case. As the steps outline, the function continues to call removeTuples until it reaches the base case, at which point it performs all necessary concatenations, returning a list filtered for duplicates.

Passing pocketTwos into removeTuples returns our desired result – the six possible combinations of pocket twos:

[("2c","2d"),("2c","2h"),("2c","2s"),("2d","2h"),("2d","2s"),("2h","2s")]

Assuming we are in a poker game dealt by some sort of mechanic or genie, we can easily return all combinations of pocket twos with replacement. We simply have to drop the condition that x does not equal y from our initial list comprehension.

twos = [(x,y) | x <- [["2c", "2d", "2h", "2s"], y <- ["2c", "2d", "2h", "2s"]]

twosReplacement = removeTuples twos

Combinations of Ace-King

Our final act – returning all combinations of Ace-King – is anti-climactic. Because our tuples comprise elements from different lists, we arrive at our answer with a single list comprehension.

bigSlick = [(x,y) | x <- ["Ac","Ad","Ah","As"], y <- ["Kc","Kd","Kh","Ks"]]

Regardless, passing this comprehension to removeTuples will have no undesired effects. No removals will take place.