// Homework #1 By Dane E. Parchment Jr. | 4925790 | Teammate: Luis Averhoff

// First three problems by Luis Averhoff the remaining by Dane Parchment

// 1.) Write an uncurried F# function cartesian (xs, ys) that takes as input two lists xs and ys and returns a list of pairs that represents the Cartesian product of xs and ys. // (The pairs in the Cartesian product may appear in any order.).

let rec cartesian = function

| (xs, []) -> []

| ([], ys) -> []

| (x::xs, ys) -> (List.map(fun y -> x,y) ys) @ (cartesian(xs,ys))

// WRITEUP

// This problem was a relatively straightforward one.

// First we needed to figure what a cartesian product was, and once that was done we needed to figure out how to do it recursively.

// A cartesian product is basically just the set that contains all the possible ordered pairs for two sets.

// In order to implement this we decided to use a pattern matching function approach rather than going for an accumulator, which in

// hindsight may have been easier to implement.

//

// We start by determining the three main cases that we will have:

// Case1: Any of the lists are empty, in which case we return an empty set, as that is the cartesian product of a set to an empty one.

// Case2: Both sets have elements in them in which case we:

// First break out the first item of one of the lists, in our case the leftmost list xs

// Next we iterate through the rightmost list, in our case ys, using the map function

// Next we create take each value from the ys list, and create a new tuple with the x that we removed earlier: (x, y)

// We then append this list of tuples into the new list created by performing a recursive call on the now declining lists.

// BUGS

// None that I could think of

//2.) An F# list can be thought of as representing a set, where the order of the elements in the list is irrelevant. Write an F# function powerset such that powerset set //returns the set of all subsets of set.

let rec powerset = function

| [] -> [[]]

| (x::xs) -> let xss = powerset xs

List.map(fun xs -> x::xs) xss @ xss

// WRITEUP

// This second problem was intrisicely much easier to solve than the first one. Like the first problem we needed to figure out

// what a powerset was, and then how to implement it recursively.

//

// A powerset is basically the set of all ordered pairs within a single set, including the empty set.

// To implement something like that we first had to figure out the cases:

// Case 1: The set is empty, in which case we return a new list that contains an empty list within it, much like the P(empty): {empty}

// Case 2: The set has elements in it, in which case:

// We first remove the head of the list.

// What we want to do is append the element to do is loop through the original list and create new lists out of the head element, and

// the elements already found in the list. However, a problem arises because if we are removing items from the list via ::, then the

// list will shrink as we try to create new lists out of it, and we would eventually run out of elements before we could finish the

// powerset.

//

// So we came up with a trick to split the list into two parts, and generate the powerset of both halfs, then append them together.

// This is accomplished by creating a new list xss that is equal to the powerset of the remaining xs that is left over after removing

// the head in each iteration. At the end we just append the two lists created to eachother which gives us the powerset!

//BUGS

// I am not to sure how fast this solution is, you mentioned how tail recursion could be faster, but I don't think we are using tail recursion

// here. So this may be slow for larger lists

//3.) The transpose of a matrix M is the matrix obtained by reflecting Mabout its diagonal.

//Write an efficient F# function to compute the transpose of an m-by-nmatrix:

let rec transpose matrix = match matrix with

| row::rows -> // When the row is not empty

match row with

| col::cols-> // When column is not empty

let first = List.map List.head matrix // Get all the elements from all rows in the list of lists

let rest = transpose(list.map List.tail matrix) // Transpose the remaining elements.

first::rest

| \_ -> [] // column empty

| \_ -> [] // row empty

// WRITEUP

// While not the most difficult this problem was probably one of the longest.

// I will keep this one brief:

// First we create a recursive function that accepts a matrix. We then match this matrix with the rows and columns that can be found in it

// and as long as a row and column exists we get all the elements within the row, and then transpose the remaining items.

// Then we append the rows to the columns, which will give us the transpose when done recursively.

// Finally we return empty lists if the columns or rows are empty

//4. ) In this problem and the next, I ask you to analyze code, as discussed in the last section of the Checklist. Suppose we wish to define // an F# function to sort a list of integers into non-decreasing order. For example, we would want the following behavior:

let rec sort = function

| [] -> []

| [x] -> [x]

| x1::x2::xs -> if x1 <= x2 then x1 :: sort (x2::xs)

else x2 :: sort (x1::xs)

// Answer and Writeup

// Let's evaluate the list by looking at all of the checklist items

// 1. Make sure that each base case returns the correct answer.

// - The application seems to be returning the correct inputs at the specified base cases. The only one that is debatable is

// whether or not the empty case should return an empty list, or fail. I do believe the that the sort of an empty list is an empty

// list though, so I guess it would depend on the point of your program. In some cases the empty case should fail, especially if

// the list is going to be used later in the program if it is assummed to have been sorted correctly.

// 2. Make sure that each non-base case returns the correct answer, assuming that each of its recursive calls returns the correct answer.

// - I believe the application will pass in this case as we check whether or not x1 is less than or equal to x2 in order to determine

// who gets :: first. If it is greater than then x2 gets added first, otherwise x1 will be the first element in the new list.

// 3. Make sure that each recursive call is on a smaller input.

// - This case will also pass because the list is constantly having an element removed from it before it is sorted recursively again.

// even though one of the elements is readded to the list during the sort, an element is still removed, so the list has n - 1 element

// per iteration.

// Conclusion: This function follows all the cases for checklist, so it passes!

//5. ) Here is an attempt to write mergesortin F#:

let rec merge = function

| ([], ys) -> ys

| (xs, []) -> xs

| (x::xs, y::ys) -> if x < y then x :: merge (xs, y::ys)

else y :: merge (x::xs, ys)

let rec split = function

| [] -> ([], [])

| [a] -> ([a], [])

| a::b::cs -> let (M,N) = split cs

(a::M, b::N)

// let rec mergesort = function

// | [] -> []

// | L -> let (M, N) = split L

// merge (mergesort M, mergesort N)

// Answer and Writeup

// We assume that split and merge work as expected, however, we must check and see if the mergesort follows the recursion checklist!

// 1. Make sure that each base case returns the correct answer.

// - For the two cases provided the mergesort is definitely returning the correct answers. However, their seems to be a case missing, I am

// uncertain if I am supposed to take of points here for that, so I will consider the program functionally incomplete, and as such does

// not pass the first checklist item.

// 2. Make sure that each non-base case returns the correct answer, assuming that each of its recursive calls returns the correct answer.

// - The problem here is type inference, since we are missing a base case, our mergesort returns a type of: a' list -> b' list, when it

// should be returning: a' list -> a' list, since it is the same list just being sorted. So I will also consider this part of the

// checklist a failure as well.

// 3. Make sure that each recursive call is on a smaller input.

// - This go round the function is fine here, since the list is split we are working with smaller inputs than the original size of the

// list provided, so this part passes.

// Conclusion: This does not follow the checklist properly and will have to be fixed, see the fix below:

let rec mergesort = function

| [] -> []

| [a] -> [a] //Fixed here

| L -> let (M, N) = split L

merge (mergesort M, mergesort N)

// By providing the base case fix, above we keep the list as returning a' list -> a' list, and as such fix the 2nd item in the checklist.

// Likewise this base case returns the proper answer, so the first checklist item also passes, meaning that mergesort now passes the

// the checklist!

//6. ) To this end, define an F# function curry f that converts an uncurried function to a curried function, and an F# function uncurry f that // does the opposite conversion.

let curry f = (fun x -> fun y -> f(x,y))

let uncurry f = (fun (x,y) -> f x y)

// WRITEUP

// This was probably the easiest part of the assignment (assumming we did this correctly). Here we needed to create a function that curries

// and a function that uncurries. A curried function basically takes its input which is a function that accepts a tuple of parameters. Into

// a function that accepts the same parameters as a chain of functions that each accept one argument each! This is reflected in the type

// CURRY TYPE: a:('a \* 'b -> 'c) -> x:'a -> y:'b -> 'c

// An uncurried function basically the complete opposite of a curried function, it take the input that is a chain of functions and transforms

// them into a single function that accepts the same parameters as a tuple. This is also reflected by its type:

// UNCURRY TYPE: a:('a -> 'b -> 'c) -> x:'a \* y:'b -> 'c

// To program these I just made functions that took a function as a parameter and then returned the same type reflected above, respectively of

// course.

// BUGS

// NOne that I could think of!