

```
// -----
//      Homework 2
//      Adam Levy, Alejandro Palacios, & Alicia Rodriguez
//      October 3, 2016
// -----
```

// 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.) For example,

```
(*
  > cartesian (["a"; "b"; "c"], [1; 2]);;
  val it : (string * int) list =
    [("a", 1); ("b", 1); ("c", 1); ("a", 2); ("b", 2); ("c", 2)]
*)
```

// Solution:

```
(*
  Cartesian
  This function calculates the cartesian product using the map function. The map
  function in the non-recursive case will create a list of tuples where the head
  of the first list (from the parameters given) is the first component of the
  tuple and the second component is each of the elements of the second list.
  (E.g: For the first call it would return [(a,1),(a,2)] with the example
  given).
  The function is then called recursively, removing the head of the first list
  since we already have all the tuples needed with that element. The base case
  will simply return an empty list once all of the elements from the first list
  are removed.
```

```
*)
let rec cartesian = function
| ([], _) -> []
| (x::xs, ys) -> List.map (fun zs -> (x,zs)) ys @ cartesian(xs, ys)
```



// 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. For example,

```
(*  
  > powerset [1;2;3];;  
  val it : int list list  
  = [[]; [3]; [2]; [2; 3]; [1]; [1; 3]; [1; 2]; [1; 2; 3]]  
*)
```

// Note that you can order the elements of the powerset however you wish.

// Solution:

```
(*  
  Powerset  
  This function first checks if the list is empty and then returns the empty  
  set. Then checks if there's only one element in the list and returns that  
  element plus the empty set.  
  Then in the recursive case, when there's more than one element on the list,  
  it assumes that the powerset of xs returns the correct input; therefore,  
  when mapping through each element of the powerset of xs, it will cons x in  
  the beginning of each element of xs.  
*)  
let rec powerset = function  
| [] -> [[]]  
| [x] -> [x] :: [[]]  
| x::xs -> List.map (fun a -> x::a) (powerset xs) @ powerset xs
```

Your second base case is unnecessary.

Also, you should use "let" to avoid doing the recursive call twice.

// 3. The transpose of a matrix M is the matrix obtained by reflecting M about  
// its diagonal. For example, the transpose of

```
(*  
  / 1 2 3 \  
  \ 4 5 6 /  
*)
```

// is

```
(*  
  / 1 4 \  
  | 2 5 |  
  \ 3 6 /  
*)
```

// An m-by-n matrix can be represented in F# as a list of m rows, each of which  
// is a list of length n. For example, the first matrix above is represented as  
// the list

```
(*  
  [[1;2;3];[4;5;6]]  
*)
```

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

```
(*  
  > transpose [[1;2;3];[4;5;6]];;  
  val it : int list list = [[1; 4]; [2; 5]; [3; 6]]  
*)
```

// Assume that all the rows in the matrix have the same length.

// Solution:

```
(*  
  Transpose  
  This function gets the head of each list within the list list and conses it  
  to the recursive call of the transpose function that will take the tail of  
  each of the lists within the list list. The base case matches the pattern  
  where the first list within the list list is empty (if the first list is  
  empty, they all are).  
  The second handles any other case. It first gets the head of each inner list
```

by mapping `list.head` onto the input list. That is consed onto the recursive call which gets the tail of each inner list in the main list by calling the `map` function.

\*)

```
let rec transpose = function
| [] :: xs -> []
| xs -> (List.map (List.head) xs) :: transpose(List.map (List.tail) xs)
```

Actually, `transpose []` should be an exception, since you cannot transpose a 0-by-k matrix.

// 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:

```
(*  
  > sort [3;1;4;1;5;9;2;6;5];;  
  val it : int list = [1; 1; 2; 3; 4; 5; 5; 6; 9]  
*)
```

// We might try the following definition:

```
let rec sort = function  
| []      -> []  
| [x]     -> [x]  
| x1::x2::xs -> if x1 <= x2 then x1 :: sort (x2::xs)  
                else x2 :: sort (x1::xs)
```

// Analyze the correctness of this definition with respect to the Checklist for  
// Programming with Recursion, being sure to address all three Steps.

// Solution:

// Analyzing Code with the Checklist

//

// 1. Is there any circumstance in which a base case fails to return the  
// correct result for the input?  
// > No.

// 2. Is there any circumstance in which the code for a non-base case can fail  
// to transform correct results returned by the recursive calls into the  
// correct result for the input?

// > It is only sorting pairs of elements because it assumes that the list  
// > it creates is already sorted. Therefore, it fails on this step.  
// > This can be seen when going over the code by hand:

```
(*  
  x1 = 3  
  x2 = 1  
  xs = [4;1;5;9;2;6;5]  
  
  [1]
```

The explanation of the Step 2 failure needs to say why the non-base case can return the wrong answer even when the recursive call returns the correct answer.

It is not enough to show that the code does not work on some input!

```
x1 = 3
x2 = 4
xs = [1;5;9;2;6;5]
```

```
[1;3]
```

```
x1 = 4
x2 = 1
xs = [5;9;2;6;5]
```

```
[1;3;1]
```

```
> It is not sorting correctly at this point.
```

```
*)
```

```
// 3. Is there any circumstance in which the definition can make a recursive
```

```
// call on an input that is not smaller than the original input?
```

```
// > No.
```

// 5. Here is an attempt to write mergesort in 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)
```

// 1. Analyze mergesort with respect to the Checklist for Programming with  
// Recursion, again addressing all three Steps. (Assume that merge and split  
// both work correctly, as indeed they do.)

// 1. Is there any circumstance in which a base case fails to return the correct  
// result for the input?  
// > No.


// 2. Is there any circumstance in which the code for a non-base case can fail  
// to transform correct results returned by the recursive calls into the correct  
// result for the input?  
// > No.

// 3. Is there any circumstance in which the definition can make a recursive  
// call on an input that is not smaller than the original input?  
// > Yes, M could be a bigger input than N when it is being split for the case  
// > that L is just a list of size one.

M being bigger than N is irrelevant.  
What matters is that M might not be shorter than L.

// 2. Enter this program into F# and see what type F# infers for mergesort.  
// Why is this type a clue that something is wrong with mergesort?


// Mergesort Type



```
(*  
  val mergesort : _arg1:'a list -> 'b list when 'b : comparison  
*)
```

// > The type says that mergesort takes in an 'a list and returns a 'b list  
// > when 'b is of a comparison type. It should be of an 'a list and not a 'b  
// > list.

// 3. Based on your analysis, correct the bug in mergesort.



```
let rec mergesort = function  
| [] -> []  
| [x] -> [x]  
| L -> let (M, N) = split L  
        merge(mergesort M, mergesort N)
```

// New Mergesort type

```
(*  
  val mergesort : _arg1:'a list -> 'a list when 'a : comparison  
*)
```



// 6. Recall that an F# function that takes two arguments can be coded in either  
 // uncurried form (in which case it takes a pair as its input) or curried form  
 // (in which case it takes the first argument and returns a function that takes  
 // the second argument). In fact it is easy to convert from one form to the  
 // other in F#. 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. For example,

```
(*
  > (+);;
  val it : (int -> int -> int) = <fun:it@13-7>
  > let plus = uncurry (+);;
  val plus : (int * int -> int)
  > plus (2,3);;
  val it : int = 5
  > let cplus = curry plus;;
  val cplus : (int -> int -> int)
  > let plus3 = cplus 3;;
  val plus3 : (int -> int)
  > plus3 10;;
  val it : int = 13
*)
```

// Curry function

```
let curry a = (fun x -> fun y -> a(x,y))
```

// OR

```
let curry f x y = f (x,y)
```

// Curry Type

```
(*
  val curry : a:('a * 'b -> 'c) -> x:'a -> y:'b -> 'c
  > This means it takes in a function (which is evaluated as curried in the end)
  > and two inputs. Both functions return the same type.
*)
```

// Uncurry function

```
let uncurry a = (fun (x,y) -> a x y)
```



// OR

```
let uncurry f (x,y) = f x y
```

// Uncurry Type

(\*

```
val uncurry : a:( $\alpha \rightarrow \beta \rightarrow \gamma$ )  $\rightarrow$  x: $\alpha$  * y: $\beta$   $\rightarrow$   $\gamma$ 
```

> This means it takes in a function (which is evaluated as uncurried in the  
> end) and two inputs. Both functions return the same type.

\*)