Part B: Structural and generative recursion (20 points)

In the lectures, we talked about the difference between structural and generative recursion. In this section we'll give you a chance to learn about generative recursion first-hand, by implementing a version of a sorting algorithm called "quicksort". The version we'll implement may be quite different from ones you may have seen before; this version of quicksort will work on lists (not arrays) and will not change the input list. This section also includes some other problems involving structural and generative recursion.

In SICP, the authors introduce the `filter` higher-order function. Translated into OCaml, that definition would be:

```
let rec filter predicate sequence =
  match sequence with
    | [] -> []
    | h :: t when predicate h -> h :: filter predicate t
    | _ :: t -> filter predicate t
```

where `predicate` is a function of one argument returning a `bool`, and `sequence` is a list. We will use this in this section. (In the OCaml libraries, this function is available as `List.filter`).

1. Quicksort

[8 points]

Implement a `quicksort` function that sorts a list of integers in ascending order, returning the new (sorted) list. The `quicksort` function works like this:

1. If the list is empty, return the empty list.
2. Otherwise, the first element in the list is called the *pivot*. Use it to create a list of all the elements in the original list which are smaller than the pivot (using the `filter` function), and another list of elements in the original list which are equal to or larger than the pivot (not including the pivot itself). Then recursively quicksort those two lists and assemble the complete list using the OCaml list append operator (`@`).

3. To make this function extra-general, instead of using the `<` operator to define whether an element is smaller than another, abstract it around a comparison function `cmp` which takes two values and returns a `bool`. We saw examples of this in lecture 9. Make the comparison function the first argument of `quicksort`. Using `(<)` as this argument makes the function sort in ascending order (the most usual way of sorting).

Examples

```
# quicksort (<) [] ;;
- : 'a list = []
# quicksort (<) [1] ;;
- : int list = [1]
# quicksort (<) [1; 2; 3; 4; 5] ;;
- : int list = [1; 2; 3; 4; 5]
# quicksort (<) [5; 4; 3; 2; 1; 2; 3; 4; 5] ;;
- : int list = [1; 2; 2; 3; 3; 4; 4; 5; 5]
# quicksort (>) [5; 4; 3; 2; 1; 2; 3; 4; 5] ;;
- : int list = [5; 5; 4; 4; 3; 3; 2; 2; 1]
```

Our solution is 7 lines long.

2. Quicksort's recursion class

[2 points]

Explain (in an OCaml comment) why the `quicksort` function is an instance of generative recursion and not structural recursion.

3. Merge sort base cases

[5 points]

Ben Bitfiddle doesn't understand why the `merge_sort` function in the lectures has to have two base cases. He writes a version which only checks for the empty list, not for lists of length 1. Recall that the `merge_sort` function and its helper functions were defined as:

```
let rec odd_half a_list =
  match a_list with
    | [] -> []
    | [x] -> [x]  (* copy 1-element list *)
```

```
    | h :: _ :: t -> h :: odd_half t (* skip second element in list *)

let even_half a_list =
  match a_list with
    | [] -> []
    | _ :: t -> odd_half t

let rec merge_in_order list1 list2 cmp =
  match (list1, list2) with
   | ([], _) -> list2
   | (_, []) -> list1
   | (h1 :: t1, h2 :: _) when cmp h1 h2 ->
       h1 :: merge_in_order t1 list2 cmp
   | (_, h2 :: t2) ->
       h2 :: merge_in_order list1 t2 cmp

let rec merge_sort a_list cmp =
  match a_list with
    | []
    | [_] -> a_list
    | _ ->
      let eh = even_half a_list in
      let oh = odd_half a_list in
        merge_in_order
          (merge_sort eh cmp)
          (merge_sort oh cmp) cmp
```

Ben's version is identical, except for the `merge_sort` function, which looks like this:

```
let rec merge_sort a_list cmp =
  match a_list with
    | [] -> []
    | _ ->
      let eh = even_half a_list in
      let oh = odd_half a_list in
        merge_in_order
          (merge_sort eh cmp)
          (merge_sort oh cmp) cmp
```

Explain in an OCaml comment why this won't work. *Hint:* Try it on some very simple test cases.

4. Insertion sort: recursive process version

[5 points]

The insertion sort function defined in the lectures generated a linear iterative process when run. A much shorter insertion sort can be written as a linear recursive process. Fill in the `<??>` section in the following code to write the linear recursive insertion sort. Is this an example of structural

recursion or generative recursion? Write a comment explaining which kind of recursion this represents.

```
let rec insert_in_order cmp new_result a_list =
  match a_list with
    | [] -> [new_result]
    | h :: t when cmp new_result h -> new_result :: a_list
    | h :: t ->  h :: insert_in_order cmp new_result t

let rec insertion_sort cmp a_list =
  match a_list with
    | [] -> []
    | h :: t -> <??>
```

You only need to add a single line in the indicated position.