1. Use the four-step design methodology to implement a function called `find_n_times` which, given a positive integer `n` and a list, returns an `option` indicating the first element of the list that appears at least `n` times, or `None` if no element appears at least `n` times.

For example, `find_n_times 3 [1; 1; 2; 2; 2]` should return `Some 2`.

(a) Step 1 is *understanding the problem*. You don’t have to write anything for this part—you answers below will demonstrate whether or not you succeeded with Step 1.

(b) Step 2 is *formalizing the interface*. Write down the *type* of the `find_n_times` function as you might find it in a `.mli` file or module interface.

```ml
val find_n_times :
```
(c) Step 3 is *writing test cases*. Complete the following three tests with the expected behavior. We have done the first one for you, based on the problem description.

Note that some test cases are better than others, and credit will be assigned accordingly: make sure your tests cover a sufficiently broad range of “interesting” input numbers and lists. Fill in the description string of the `run_test` function with a short explanation of *why* the test case is interesting. Your description should not just restate the test case, e.g. “`find_n_times 3 [1; 1; 2; 2; 2]`”.

i.  
   ```ocaml
   let test () : bool = 
   begin match find_n_times 3 [1; 1; 2; 2; 2] with 
   | None -> false 
   | Some v -> v = 2 
   end 
   ;; run_test "find_n_times on simple list" test
   ```

ii.  
    ```ocaml
    let test () : bool = 
    begin match find_n_times with 
    | None -> 
    | Some v -> 
    end 
    ;; run_test "" test
    ```

iii.  
     ```ocaml
     let test () : bool = 
     begin match find_n_times with 
     | None -> 
     | Some v -> 
     end 
     ;; run_test "" test
     ```

(d) Step 4 is *implementing the program*. Fill in the body of the `find_n_times` function to complete the design.

Hint: You can also define a helper function.

```ocaml
let rec find_n_times (n: ____ ) (l: _________ ) : _________ =
```