CS51 **Spring 2024**

Code Review 6: Mutable State and Imperative Programming

1 Practice Problems

Problem 1.1 (Generators).

1. Define a function `mem_factorial` which takes a unit as an argument and returns 1 the first time it is called, returns 2 the second time it is called, returns 6 the third time it is called, and so on. In general, the function should return n! on the nth time it is called.

```
let mem_factorial : unit -> int =
  let n = ref 0 in
  let fact = ref 1 in
  fun () : int ->
    let prev = if !n = 0 then 1 else !fact * (!n + 1) in
    n := !n + 1;
    fact := !n * !fact;
  prev ;;
```

2. Define a function that generates the Fibonacci numbers the ith time the function is called.

   Recall that $F_1 = 1$, $F_2 = 1$, and $F_i = F_{i-1} + F_{i-2}$ for all $i > 2$.

```
let fib : unit -> int =
  let first = ref 1 in
  let second = ref 1 in
  fun () : int ->
    let ret = !first in
    let num = !first + !second in
        first := !second;
        second := num;
        ret  ;;
```

Problem 1.2 (Tail Recursion).

1. Define a non-recursive implementation of `List.filter`

```
let filter (pred : 'a -> bool) (lst : 'a list) : 'a list =
  let accum = ref [] in
  let curr_list = ref lst in
  while !curr_list != []
  do
    match !curr_list with
    | [] -> failwith "there's a bug in filter"
    | h :: t -> if pred h then accum := !accum @ [h]; curr_list :=
       t
  done;
```

```
  !accum
;;
```

2. `range n m` returns a list that contains all the integers from `n` to `m-1` inclusive. Define three versions of the `range` function: (1) a tail-recursive version, (2) a non-tail recursive version, and (3) a procedural implementation (using loops).

```
let rec range (n : int) (m : int) : int list =
if n = m then []
else n :: range (n + 1) m ;;

let range (n : int) (m : int) : int list =
  let rec range_inner (n' : int) (m' : int) (acc : int list) :
     int list =
  if n' = m' then acc
  else range_inner (n' + 1) m (acc @ [n'])
in range_inner n m [] ;;

let range (n : int) (m : int) : int list =
let acc = ref [] in
for curr = n to m
do
  acc := !acc @ [curr]
done;
!acc ;;
```

3. Discuss the advantages and disadvantages between a tail-recursive, non-tail-recursive, and iterative approaches.

Problem 1.3 (Mutable List Practice)**.**

1. Define a function that merges two sorted (in ascending order) mutable lists. This function should return a value of type unit and the first list should become the merging of the two lists.

```
let rec merge (x : 'a mlist) (y : 'a mlist) : unit =
  match !x, !y with
  | Nil, Nil -> ()
  | Nil, _ -> x := !y
  | _, Nil -> ()
  | Cons (hx, tx), Cons (hy, ty) ->
    if hx < hy then merge tx y
    else (merge ty x; x := Cons(hy, ty)) ;;
```

Problem 1.4 (Big-O)**.** State which complexity classes (listed in proposition 5.5) that the following functions fall under.

1. $f(n) = 3n^2 + 10n \log n + 1000n + 4 \log n + 9999$

> **Solution**
> Quadratic, Cubic, Exponential

2. $f(n) = (\log n)^2 + 2n + 4n + \log n + 50$

> **Solution**
> Linear, LogLinear, Quadratic, Cubic, Exponential. The tighest complexity class is Linear, which means that $f(n) \in O(n)$. Why is this the case?
>
> Focus on the $(\log n)^2$ term. $\log n \in O(\sqrt{n})$, so $O(\log n)^2 \in O(\sqrt{n})^2 = O(n)$.

3. $f(n) = n + n^2 + 2n + n^4$

> **Solution**
> Exponential

4. $f(n) = e^n + 5^{\log n}$

> **Solution**
> Exponential

Problem 1.5 (Power)**.**

1. Consider the following implemention of `power`.

```
let rec power (x : int) (n : int) : int =
    if n = 0 then 1
    else x * power x (n - 1) ;;
```

Define the worst-case time complexity of this function in terms of n.

> **Solution**
> The time complexity is $O(n)$

2.
```
let rec power (x : int) (n : int) : int =
    if n = 0 then 1
    else if n mod 2 = 0 then power x (n / 2) * power x (n / 2)
    else  power x (n / 2) * power x ((n / 2) + 1);;
```

Define the worst-case time complexity of this implementation in terms of n.

> **Solution**
> The worst case time-complexity is $O(n)$.
>
> The recurrence here is $T(n) = c + 2 \cdot T(\frac{n}{2})$, which by using the table recurrence patterns, $T(n) = O(n)$.

3. Rewrite `power` from Part 2 to improve the worst-case time complexity. What is the asymptotic runtime of the function right now?

```
let rec power (x : int) (n : int) : int =
  if
```

```
    n = 0 then 1
  else
    let n2_power = power x (n / 2) in
      if n mod 2 = 0 then
        n2_power * n2_power
    else
        n2_power * n2_power * x  ;;
```

Solution

The time complexity is $O(\log n)$.

Problem 1.6 (Minimum). In lab 10, we defined a function `find_min` that finds the minimum of the list.

Listing 1: find_min

```
let rec find_min (xs : int list) : int =
  let rec split (xs : 'a list) =
    | [] -> [], []
    | [x] -> [x], []
    | first :: second :: rest -> let rest1, rest2 = split rest in
                                 first :: rest1, second :: rest2 in

  match xs with
  | [] -> raise (Invalid_argument "Empty List")
  | [x] -> x
  | _ -> let xs1, xs2 = split xs in
         min (find_min xs1) (find_min xs2) ;;
```

Derive the asymptotic runtime of the `find_min` function.

Solution
The `split` helper function splits the list into halves.
$T_{\texttt{find\_min}}(1) = c$ where c is a constant.
$T_{\texttt{find\_min}}(n) = 2T_{\texttt{find\_min}}(n/2) + T_{\texttt{split}}(n) + c$ for $n > 1$.
Intuitively, we split the input `xs` into two halves (`xs1` and `xs2`), and then recursively call `find_min` on each of those two halves. `min` takes constant time because it only requires a constant number of comparisons.
Before, we proceed, we want to find $T_{\texttt{split}}(n)$ in terms of n.
`split` is also a recursive function, so we can write a recurrence.
$T_{\texttt{split}}(0) = c$; $T_{\texttt{split}}(1) = c$, and $T_{\texttt{split}}(n) = T_{\texttt{split}}(n-2) + c$.[a]
Unfolding the recurrence:

1) $T_{\texttt{split}}(n-2) = T_{\texttt{split}}(n-4) + c$
2) $T_{\texttt{split}}(n-4) = T_{\texttt{split}}(n-6) + c$
3) By substitution, $T_{\texttt{split}}(n) = T_{\texttt{split}}(n-6) + 3c$.
4) We notice a pattern that $T_{\texttt{split}}(n) = T_{\texttt{split}}(n-2k) + kc$.
5) In the base case, $n - 2k = 0$, so $n = 2k$ (the odd case doesn't really matter since the $+1$

factor is negligible for large n).

6) Thus, $T_{\texttt{split}}(n) = c + \frac{cn}{2} \in O(n)$, so $T_{\texttt{split}}(n) \in O(n)$.

By substitution, $T_{\texttt{find\_min}}(n) = 2T_{\texttt{find\_min}}(n/2) + n + c$ [b].

We can now unfold $T_{\texttt{find\_min}}(n)$:

1) $T_{\texttt{find\_min}}(n/2) = 2T_{\texttt{find\_min}}(n/4) + \frac{n}{2} + c$
2) $T_{\texttt{find\_min}}(n/4) = 2T_{\texttt{find\_min}}(n/8) + n/4 + c$
3) By substitution, $T_{\texttt{find\_min}}(n) = 8T_{\texttt{find\_min}}(n/8) + 4n + 3c$.
4) We can generalize this to the pattern: $T_{\texttt{find\_min}}(n) = 2^k T_{\texttt{find\_min}}(n/2^k) + (k+1)n + kc$
5) When we get to our base case (i.e. the recursion stops), $\frac{n}{2^k} = 1$, $n = 2^k$, and $k = \log n$.
6) By substitution, $T_{\texttt{find\_min}}(n) = 2^{\log n} + (\log n + 1)n + c \log n = n + (\log n + 1)n + c \log n \in O(n \log n)$.

Thus, $T_{\texttt{find\_min}}(n) \in O(n \log n)$.

---

[a] Note that c is not necessarily exactly the same in all three equations, so it would be more precise to say that it really takes $O(1)$ time, but since the constants won't matter in terms of determining the Big-O class, we use c in all cases for simplicity.

[b] Note that $T_{\texttt{split}}(n)$ is not exactly n, but since we're concerned with *asymptotic* runtime constant factors don't matter