COMP 212 Spring 2023
Homework 04

1 Introduction

Because the programming and written tasks are integrated, all problems are described in this handout, but there is a separate *handin* sheet for the written on the course web page. You should hand in your `hw04.sml` file and the written handin (or your own write-up, if you prefer) on Google Drive. You must write purposes and tests for all functions on this assignment (and on all future assignments!).

There is no challenge problem this week to save a little extra time for you to get used to algorithm running time analysis.

2 Prefix-Sum

The prefix-sum of a list `l` is a list `s` where the i^{th} element of `s` is the sum of the first $i+1$ elements of `l`. For example,

```
prefixSum [] ≅ []
prefixSum [1,2,3] ≅ [1,3,6]
prefixSum [5,3,1] ≅ [5,8,9]
```

Here is a simple implementation of this function:

```
fun prefixSum (l : int list) : int list =
    case l of
      [] => []
    | x::xs => x :: add_to_each (prefixSum xs, x)
```

The function `add_to_each` is the "raise salaries" function from lecture; it adds `x` to each element of `prefixSum xs`.

Task 2.1 (5 pts). Write a recurrence for the work of `prefixSum`, $W_{\texttt{prefixSum}}(n)$, where n is the length of the input list. Give a closed form for this recurrence. Give a big-O bound for $W_{\texttt{prefixSum}}(n)$.

You may use variables $k_0, k_1, \ldots$ for constants. You should assume that `add_to_each` is a linear time function: `add_to_each l` evaluates to a value in kn steps where n is the length of `l` and k is some constant; your recurrence should involve the constant k.

In order to compute the prefix sum operation faster, we will use the technique of adding an additional argument: *harder problems can be easier*.

To do this, you should write a `prefixSumHelp` function that uses an additional argument to compute the prefix sum operation in time that is asymptotically better than the above function. You must determine what the additional argument should be.

Task 2.2 (5 pts). Give a precise mathematical specification for your `prefixSumHelp` function, which describes the role of the extra argument by relating `prefixSumHelp` to `prefixSum`. You do not need to prove the relationship, but use it to help you write the code.

Task 2.3 (10 pts). Write the function. Once you have defined `prefixSumHelp`, use it to define the function

```
prefixSumFast : int list -> int list
```

that computes the prefix sum.

Task 2.4 (10 pts). Write recurrences

- $W_{\texttt{prefixSumHelp}}(n)$, for the work of `prefixSumHelp`, and
- $W_{\texttt{prefixSumFast}}(n)$, for the work of `prefixSumFast`$(n)$,

in terms of the length n of the input list. Give a closed form for both recurrences, and then give a big-O bound for both.

3 Shuffle

Consider the following function, which mixes up the order of the elements of a list (`append` is from Lecture 6, `split` is from Lecture 8):

```
fun shuffle (l : int list) : int list =
    case l of
        [] => []
      | [x] => [x]
      | _ => let val (p1, p2) = split l
             in
                 append(shuffle p1, shuffle p2)
             end
```

Task 3.1 (10 pts). Give a recurrence and big-O bound for the work of `shuffle` on an input list of length `n`. Briefly explain your answer.

4 Quicksort

In this problem, you will implement QuickSort on lists. The idea for the algorithm is as follows:

1. The empty list is sorted.
2. Any singleton list is sorted.
3. If the list being sorted has more than one element:
   (a) Pick a pivot element from the list being sorted.
   (b) Divide the list being sorted into two lists: a list of elements less than the pivot and a list of elements greater than or equal to the pivot.
   (c) Call QuickSort recursively to sort both lists.
   (d) Append these lists together with the pivot in the appropriate order.

The main concern in making QuickSort a practical algorithm comes in choosing the pivot. A naïve answer is to always choose the first element of the list for your pivot, and this is what you will implement in this assignment. This simple policy gives QuickSort a worst case work in $O(n^2)$ on a list of n elements, rather than the $O(n \log n)$ we got for `mergesort`: if the list being sorted has no repeated elements, and happens to be in reverse-sorted order, then the list of elements greater than or equal to the pivot contains only the pivot, and the list of elements less than the pivot has $(n-1)$ elements. However, quicksort has $O(n \log n)$ *expected* running time averaged over all possible lists (though we will not formally talk about expected case analysis in this class) — for "most" lists the two subproblems will be roughly balanced.

First, you will need a helper function that selects the elements less and greater than the pivot.

Task 4.1 (15 pts). Implement function

```
filter_less : int list * int -> int list
filter_greatereq : int list * int -> int list
```

`filter_less(l,p)` computes a list with all and only those elements of `l` that are less than the pivot `p`. `filter_greatereq(l,p)` computes a list with all and only those elements of `l` that are greater than or equal to the pivot `p`.[1]

Task 4.2 (15 pts). Implement QuickSort on lists in the function

```
quicksort_l : int list -> int list
```

For any list l, `(quicksort_l` l`)` should evaluate to a permutation of l that is sorted in increasing order.

[1] It's OK if there is some copy-and-paste between these two functions; we will fix this later in the course.