# Homework 5: High Order ML Functions

*Stephen Wagstaff*
*CS 431*
*October 10, 2018*

---

1. Write a function `merge_sort: ('a * 'a -> bool) -> 'a list -> 'a list` that takes a comparison function and then a list, and return a sorted list in the order specified by the comparison function.

   - For example, `merge_sort (op >) [0, 5, 1, ~4, 9, 11]` evaluates to `[11,9,5,1,0,~4]`.

```
fun merge_sort _ nil = nil
  | merge_sort f (x::nil) = x::nil
  | merge_sort f list =
  let
    fun split list =
    let
      val cut = (length list) div 2
    in
      ( List.take (list,cut), List.drop (list,cut) )
    end;

    fun merge (nil, remaining) = remaining
      | merge (remaining, nil) = remaining
      | merge ((left as lh::l), (right as rh::r)) =
      if f(lh, rh)
      then lh::merge (l, right)
      else rh::merge (left,r);
    val (left, right) = split list
  in
    merge ((merge_sort f left), (merge_sort f right))
  end;
```

2. Write a function `selection_sort: ('a * 'a -> bool) -> 'a list -> 'a list` that takes a comparison function and then a list, and return a sorted list in the order specified by the comparison function. Selection sort orders a list from the left to right by selecting the least element from the unsorted portion of the list and adds it to the partially sorted list. Note that the least element is not necessarily the smallest or largest element – it depends on the comparison function passed to selection_sort.

   - For example, `selection_sort (op >) [0, 5, 1, ~4, 9, 11]` evaluates to `[11,9,5,1,0,~4]`. The computation has the following steps:

```
    [0, 5, 1, ~4, 9, 11] select 11
->  11::[0, 5, 1, ~4, 9] select 9
->  11::9::[0, 5, 1, ~4] select 5
->  11::9::5::[0, 1, ~4] select 1
->  11::9::5::1::[0, ~4] select 0
->  11::9::5::1::0::[~4] select ~4
->  11::9::5::1::0::~4::nil
```

   You should implement a helper function `select: 'a list -> 'a list` as an inner function to selection_sort. The select function takes a list and returns the list with the least element in front. For example, `select [0,5,1,~4,9,11]` should return `[11,0,5,1,~4,9]` given the ordering function `op >`.

```
fun selection_sort _ nil = nil
  | selection_sort f list =
let
  fun select nil = nil
    | select [x] = [x]
    | select (firstElement::list) =
      let
        val (testElement::remaining) = select(list)
```

```
      in
        if(f(testElement,firstElement))
        then select(testElement::firstElement::remaining)
        else firstElement::testElement::remaining
      end;
in
  select list
end;
```

3. Write a function `insertion_sort: ('a * 'a -> bool) -> 'a list -> 'a list` that takes a comparison function and then a list, and return a sorted list in the order specified by the comparison function. Insertion sort orders a list from left to right by inserting an element from unsorted portion of the list into the correct position in the sorted portion of the list.

   - For example, `insertion_sort (op >) [0, 5, 1, ~4, 9, 11]` evaluates to `[11,9,5,1,0,~4]`. The computation has the following steps:

```
    nil [0, 5, 1, ~4, 9, 11] insert 0
->  [0] [5, 1, ~4, 9, 11] insert 5
->  [5, 0] [1, ~4, 9, 11] insert 1
->  [5, 1, 0] [~4, 9, 11] insert ~4
->  [5, 1, 0, ~4] [9, 11] insert 9
->  [9, 5, 1, 0, ~4] [11] insert 11
->  [11, 9, 5, 1, 0, ~4] nil
```

   You should implement two helper functions as inner functions. The first function `insert:'a list * 'a -> 'a list` takes a sorted list `l` and an element `x` and insert `x` into the right position of `l` so that the list remains sorted after insertion. For example, `insert ([5,1,0,~4], 9)` should evaluates to `[9,5,1,0,~4]`. Again the order depends on the comparision function passed to `insertion_sort`. The second helper function `sort:'a list * 'a list -> 'a` list takes a sorted list `sofar` and an unsorted list `l` and inserts the elements of `l` into `sofar`.

```
fun insertion_sort _ nil = nil
  | insertion_sort f list =
    let
      fun insert x nil = [x]
        | insert x (head::rest) =
          if f(x,head)
          then x::head::rest
          else head::(insert x rest)
      fun sort sofar nil = sofar
        | sort sofar (element::list) =
          insert element (sort sofar list)
    in
      sort nil list
    end;
```

4. Write a function `quicksort: (int * int -> bool) -> int list -> int list` that takes a comparison function (i.e. either `op >` or `op <`), then a list of integers, and return a sorted list of integers in the order specified by the comparison function.

   - **requirement:**
   - (a) You should implement quicksort algorithm.
   - (b) You should have three base cases: empty list, list of one, and list of two elements.
   - (c) The pivot is the average of the first, middle, and the last element of the list. You should define an inner function `get: int list * int -> int` that takes a list and return an element of the list specified by an index.
   - For example, `get(lst, 1)` returns the first element of `lst`, `get(lst, (length lst + 1) div 2)` returns the middle elment, and `get(lst, length lst)` returns the last element.
   - (d) The `split` function should split the list into three sublists: *(lower, middle, upper)*, where the middle list are elements that are equal to the pivot. The lower and upper lists are elements that are either less or greater than the pivot (depending on the comparison function).

- (e) Note that you place the middle list between the sorted lower and upper list.

```
fun quicksort (_, nil) = nil
  | quicksort (_, [x]) = [x]
  | quicksort (f, rest) =
  let
    fun get (head::_, 1) = head
      | get (_::rest, index) = get(rest, index-1);

    val pivot = (get(rest, 1) + get(rest,(length rest + 1) div 2) + get(rest, length rest) div 3)

    fun split nil = ( nil, nil, nil )
      | split (x :: remaining) =
      let
        val ( lower, middle, upper ) = split remaining
      in
        if f( pivot, x )
        then ( lower, middle, x::upper )
        else
          if x = pivot
          then (lower, x::middle, upper)
          else (x::lower, middle, upper )
      end;
    val ( lower, middle, upper ) = split rest
  in
    quicksort(f, lower) @ middle @ quicksort(f, upper)
  end ;
```