AC21007: Haskell Lecture 5
Selection Sort, Insertion Sort, and Bubble Sort

František Farka

Recapitulation

- Function type `a -> b`
- Anonymous functions
- Currying
- Higher order functions
  - `map`, `filter`
  - Folds: `foldr`, `foldl`

Selection Sort

Goal: We must devise an algorithm that sorts a collection of elements

Solution: From those elements that are currently unsorted, find the smallest and place it next in the sorted collection.

Example:

```
                       ↓
[]              [7, 5, 2, 4]
                       ↓
[2]             [7, 5, 4]
                    ↓
[2, 4]          [7, 5]
                 ↓
[2, 4, 5]       [7]

[2, 4, 5, 7]    []
```

Selection Sort - C version

- Implementation in C:

```
void sel_sort(int* a, size_t n) {

    for (size_t i = 0, j; i < (n - 1); ++i) {
        j = i;

        for (size_t k = i + 1; k < n; ++k) {
            if (a[k] < a[j])
                j = k;

            /* int t = a[i]; a[i] =a[j]; a[j] = t; */
            swap(a, i, j);
        }
    }
}
```

Selection Sort - Haskell version

Goal: . . . an algorithm that sorts a *list* of elements

Solution: . . . from unsorted, find the smallest and place it next in the sorted list. Empty list is trivially sorted!

Function:

```
selSortImpl :: [Int] -> [Int] -> [Int]
selSortImpl sorted []  = sorted
selSortImpl sorted xs  =
      selSortImpl (sorted ++ [x]) (removeFirst x xs)
    where
       x = minimum xs
       removeFirst _ [] = []
       removeFirst a (x:xs) = if x == a
          then xs
          else x : removeFirst a xs


selSort :: [Int] -> [Int]
selSort = selSortImpl []
```

Insertion Sort

Goal: The same ...

Solution: From those elements that are currently unsorted, take the *first* and place it correctly in the sorted list.

Example:

```
[]               [7, 2, 5, 4]
 ↓
[7]              [2, 5, 4]
 ↓
[2, 7]           [5, 4]
    ↓
[2, 5, 7]        [4]
    ↓
[2, 4, 5, 7]     []
```

Insertion Sort - Haskell version

Function:

```
insSortImpl :: [Int] -> [Int] -> [Int]
insSortImpl sorted []  = sorted
insSortImpl sorted (x:xs)  =
        insSortImpl (insert x sorted) xs
     where
         insert y [] = [y]
         insert y (z:zs) = if y <= z
            then y : (z : zs)
            else z : (insert y zs)

insSort :: [Int] -> [Int]
insSort = insSortImpl []
```

Syntactic intermezzo: `let ...in` expression

- We know `where` syntax
- The `let ...in` epression

$$
\begin{array}{ll}
\textbf{let} & \texttt{<pat}_1\texttt{> = <expr}_1\texttt{>} \\
 & \texttt{<pat}_n\texttt{> = <expr}_n\texttt{>}\quad \textbf{in}\ \texttt{<expr>}
\end{array}
$$

  is a "local" version – variables that are bound in patterns pat_1 to pat_n after evaluating expressions $expr_1$ to $expr_n$ are in scope in *expr*
- ...**and** in $expr_1$ to $expr_n$ – bindings may by recursive!
- The expresion has a value of *expr*.
- E.g.:

```
\ x -> let (y, z) = x in y + z
let x = 1 : x in x
```

Bubble Sort

Goal: The same . . .

Intuition: In each iteration *bubble up* the greatest element. But which one is it?

Solution: Start with the first element and bubble up as long as it is the greates so far, once we saw greater, continue with that one!
In each iteration, *one* element is placed (the greates), after n iterations - n elements placed!

Example:

[5,2,7,4] → [2,5,7,4] → [2,5,7,4] → [2,5,4,7]

[2,5,4,7] → [2,5,4,7] → [2,4,5,7] → [2,4,5,7]

[2,4,5,7] → [2,4,5,7] → [2,4,5,7] → [2,4,5,7]

[2,4,5,7] → [2,4,5,7] → [2,4,5,7] → [2,4,5,7]

Bubble Sort - Haskell version

Function:
```
bubbleSortImpl :: Int -> [Int] -> [Int]
bubbleSortImpl 0 xs  = xs
bubbleSortImpl n xs  =
        bubbleSortImpl (n - 1) (bubble xs)
     where
         bubble [] = []
         bubble (x : []) = x : []
         bubble (x : y : ys) = if x <= y
                then x : (bubble (y : ys))
                else y : (bubble (x : ys))

bubbleSort :: [Int] -> [Int]
bubbleSort xs = let n = length xs
    in bubbleSortImpl n xs
```

Time complexity

- Not that easy as with Turing Machine, RAM, or C
- Abstract, non-mutable structures, no (out-of-box) direct indexing:
  - In C, for `ar` array, and `n` index

    ```
    ar[n]
    ```

    is a "primitive" action, $\mathcal{O}(1)$!
  - In Haskell, for `lst` list, and `n` index

    ```
    lst !! n
    ```

    is a function call to

    ```
    (!!) :: Int -> [a] -> a
    (x:_)  !! 0  = x
    (_:xs) !! i  = xs !! (i - 1)
    ```

    in time $\mathcal{O}(n)$

Time complexity

- Not that easy as with Turing Machine, RAM, or C
- Lazy evaluation
  - In C

    ```
    int dummy_minimum(int* ar, size_t n)
    {
        sel_sort(ar, n); // runs in O(n^2)
        return arr[0];   // runs in O(1)
    }
    ```

    runs in $\mathcal{O}(n^2)$
  - In Haskell

    ```
    dummyMinimum :: [Int] -> Int
    dummyMinimum xs =
        head (              -- runs in O(1)
            selSort xs      -- only first selection
                            --   evaluated - in O(n) !
        )
    ```

    runs in $\mathcal{O}(n)$

Time complexity

- Not that easy as with Turing Machine, RAM, or C
- Abstract, non-mutable structures, no (out-of-box) direct indexing
- Lazy evaluation
- Some algorithms are naturally imperative, other are functional!

Next time

- Monday the the 15th of February, 2-3PM, Dalhousie 3G05 LT2
- Defined data types
- Ad-hoc polymorphism: Typeclasses