Sorting with Haskell

Oliver Ray

Week 10

This worksheet investigates various sorting algorithms in Haskell and looks at their best and worst case run-time complexity in terms of the number of comparisons they perform as a function of input size.

1. Write a function `partition` that given a Boolean test `t` and a list `xs` returns two lists containing the elements of `xs` that do and don't satisfy test `t`, respectively. Thus `partition even [1,2,3] = ([2],[1,3])`. Write this function in such a way that it examines each member of the list `xs` exactly *once*.

2. Write a function `qSort` that quick sorts a list of integers by using `partition` to replace the two list comprehensions given in the `qSort` function from the lecture slides. Explain any savings in efficiency.

3. Determine the number of comparison operations performed by `qSort` in the best and worst cases in terms of the length n of its input. State the order of best and worst case scenarios in Big-Oh notation.

4. Determine the number of comparison operations performed by the `iSort` insertion sort from the lecture slides in the best and worst cases. State the order of best and worst case scenarios in Big-Oh notation.

5. Write a function `bSort` that bubble sorts a list of integers according to the following description of the algorithm:

   - A bubble sort involves several passes through the input list.
   - Each pass proceeds from the front of the list towards the back.

- During the pass, consecutive pairs of integers are compared in turn and swapped if they are out of order.
- After each pass one element will have been correctly placed at the end of the list and not be examined no further.
- The sort ends when there is a pass in which no swaps are needed.

For example, the following illustration shows how the list [5,1,4,2,8] would be bubble sorted. Elements being compared are shown in bold. Elements fixed in place are shown in italics.

- start:
  5 1 4 2 8
- pass 1:
  5 1 4 2 8
  1 **5 4** 2 8
  1 4 **5 2** 8
  1 4 2 **5 8**
- pass 2:
  1 4 2 5 *8*
  1 **4 2** 5 *8*
  1 2 **4 5** *8*
- pass 3:
  1 2 4 *5 8*
  1 **2 4** *5 8*
- done:
  1 2 4 5 8

6. Determine the number of comparison operations performed by `bSort` in the best and worst cases. State the order of best and worst case scenarios in Big-Oh notation.

7. Given that the worst case performance of `qSort` is the same as both `bSort` and `iSort` and the best case performance of `qSort` is worse than both `bSort` and `iSort`, suggest why `qSort` is in fact a popular sorting algorithm in practice.

ANSWERS

1. ```
   partition :: (a->Bool) -> [a] -> ([a],[a])
   partition _ [] = ([],[])
   partition t (x:xs)
       | t x       = (x:ys, zs)
       | otherwise = (ys, x:zs)
       where (ys,zs) = partition t xs
   ```

2. ```
   qSort :: [Int] -> [Int]
   qSort [] = []
   qSort (x:xs) = (qSort ls) ++ [x] ++ (qSort gs)
       where (ls,gs) = (partition ((>=) x) xs)
   ```

   `partition` is more efficient than two comprehensions because it only goes through the list once instead of twice and thus performs half as many comparison operations.

3. Quick Sort

   Worst case (when list is sorted in ascending/descending order):

   $(n-1)+(n-2)+\cdots+1 = n^2/2 = O(n^2)$

   Best case (when $n = 2^k - 1$ and pivot value is always the median):

   $2^k(k-2)+2 = (n+1)(log_2(n+1)-2)+2 = O(n\,log\,n)$

4. Insertion Sort

   Worst case (when list is sorted in descending order):

   $1+2+\cdots+(n-1) = n^2/2 = O(n^2)$

   Best case (when list is sorted in ascending order):

   $n-1 = O(n)$

5. ```haskell
   bSort :: [Int] -> [Int]
   bSort xs = bub [] xs [] False

   bub :: [Int] -> [Int] -> [Int] -> Bool -> [Int]
   bub [] []  zs _ = zs
   bub xs [y] zs True = bub [] xs (y:zs) True
   bub xs [y] zs False = xs++(y:zs)
   bub xs (y1:y2:ys) zs b
       | y1 <= y2  = bub (xs++[y1]) (y2:ys) zs b
       | otherwise = bub (xs++[y2]) (y1:ys) zs True
   ```

6. Bubble Sort

   Worst case (when list is sorted in descending order):

   $(n-1)+(n-2)+\cdots+1 = n^2/2 = O(n^2)$

   Best case (when list is sorted in ascending order):

   $n-1 = O(n)$

7. It can be shown the average case performance of `qSort` is $n\,log\,n$ (the same as its best case) while the average case performance of both `bSort` and `iSort` is n^2 (the same as their worst cases). In other words, the worst case performance of `qSort` is only seen very rarely.
   ```