Sorting

Sorting Sequences

- The need for sorting a collection/sequence is ever present in computing
  - sort names in alphabetical order
  - sort items according to price
  - sort files according to size or date written/modified etc
- A very much studied problem
  - resulting in a large number of different sorting algorithms
  - different assumptions and properties
  - not so often implemented in practice
- Everyone should be know about
  - different sorting algorithms,
  - their differences in terms of assumptions and properties
  - how to implement a number of them
  - .. and then rely on the builtin implementation
- A good starting point for study and understanding of other problems

Sorting - The Problem

- Without loss of generality, we reduce this to sorting a sequence of integers
- So, given a sequence `[10, 1, -4, 8, 19, 17, 12]` we want to return `[-4, 1, 8, 10, 12, 17, 19]`
- In other words we want to return a *ordered permutation* of the input
  - with *ordered* we mean $e_0 \leq e_1 \leq .. \leq e_n$
    - we allow repeated elements
  - with permutation we mean containing the same elements in another sequence (repeated elements should be the same as well).
- Nice exercise: given a list, generate all permutations of that list
- We *could* sort a list be generating all permutations and then using filter to find the single one that is ordered.
  - Given that there are N! permutations, this would be *very* slow, even with lazy evaluation.

Sorting

$\boxed{\text{Think!}}$

Solutions

- Recursive solution based on
  - sorting tail recursively and inserting head into sorted tail
  - using a sorted prefix as an accumulator and inserting each element into the sorted prefix
- The latter is very much how you would sort cards from one hand to the each, i.e., by inserting a new card where it belongs
- The heart of this algorithm is a function that inserts a new element into an ordered list
- `insert :: Integer -> [Integer] -> [Integer]`
- Good exercise: implement `insert` and the whole sorting function
- This called *insertion sort*

```
Input                           Accumulator
[10, 1, -4, 8, 19, 17, 12] []
    [1, -4, 8, 19, 17, 12] [10]
       [-4, 8, 19, 17, 12] [1, 10]
           [8, 19, 17, 12] [-4, 1, 10]
              [19, 17, 12] [-4, 1, 8, 10]
                  [17, 12] [-4, 1, 8, 10, 19]
                      [12] [-4, 1, 8, 10, 17, 19]
                        [] [-4, 1, 8, 10, 12, 17, 19]
```

Solutions

- Potential alternative is to select the *largest* element and push that to the accumulator
- Adding to the accumulator is easier – just use `:`
- Finding and removing the largest element is more complex
  - when doing this keeping the order of the input is *not* important
  - the trace below is an example
- Again, good exercise to implement this
- This is called *selection sort*
- Compare with insertion with regards to where the complexity is

```
Input                           Accumulator
[10, 1, -4, 8, 19, 17, 12] []
    [10, 1, -4, 8, 17, 12] [19]
        [10, 1, -4, 8, 12] [17, 19]
            [10, 1, -4, 8] [12, 17, 19]
                [1, -4, 8] [10, 12, 17, 19]
                   [1, -4] [8, 10, 12, 17, 19]
                      [-4] [1, 8, 10, 12, 17, 19]
                        [] [-4, 1, 8, 10, 12, 17, 19]
```

Solutions

- Move through the sequence and swap adjacent pairs which are not in order
- Repeat until no swaps are done when moving through the sequence
- We see how the larger elements slowly move towards the end
  - They move like bubbles
- This is called *bubble sort*
- As expected: good exercise to implement

```
Input
[10, 1, -4, 8, 19, 17, 12]
[1, 10, -4, 8, 19, 17, 12]
[1, -4, 10, 8, 19, 17, 12]
[1, -4, 8, 10, 19, 17, 12]
[1, -4, 8, 10, 17, 19, 12]
[1, -4, 8, 10, 17, 12, 19]
[-4, 1, 8, 10, 17, 12, 19]
[-4, 1, 8, 10, 12, 17, 19]
```

Side Note

- The two first solutions, *insertion sort* and *selection sort* are recursive by moving *one* element from the input to the accumulator in each step
  - The variant is the length of the input
- They differ in terms of where the complexity and simplicity is
  - selection of element to move
  - how we add it to the accumulator
- In *bubble sort* we always have the same size of the sequence, but we rearrange the sequence
  - The variant is the "*amount of order*" - without saying what that means

Alternative Sub problems

- Instead of splitting the input by removing just *one* element, we can split the input into two (more or less) equally sized parts.
- If input is L = L1 ++ L2, we sort the parts recursively
  - S1 = sort L1
  - S2 = sort L2
- We now have two sorted lists — how do we combine them into one sorted list?
- Implement a function `merge :: [a] -> [a] -> [a]`
  - two input lists — select the smallest head in each step, keep the other list
- This is called *merge sort*
- note: splitting a list into two “equally sized” parts is not really “simple”
- fill in the details

```
Input
[10, 19, -4, 8] ++ [3, 17, 12, 1]
sort parts recursively
merge [-4, 8, 10, 19] [1, 3, 12, 17]
-4 : merge [8, 10, 19] [1, 3, 12, 17]
-4 : 1 : merge [8, 10, 19] [3, 12, 17]
-4 : 1 : 3 : merge [8, 10, 19] [12, 17]
-4 : 1 : 3 : 8 : merge [10, 19] [12, 17]
…
```

Alternative Sub problems

- Split input into one element `p` and two (more or less) equally sized parts, where the elements of the first part are `≤p` and the elements of the second part are `>p`
- We can thus see the input as three parts `prefix, p, suffix`
  - `prefix` and `suffix` are *unsorted*, but relate to `p`
- Sort the parts recursively
  - We can now just put the parts together as in
    - `sortedprefix ++ [p] ++ sortedsuffix`
- `p` is called a *pivot element*
- This is called *quick sort*
- The complexity is in selecting the pivot element and splitting the input
- Splitting is called *partitioning*
- Again, a good exercise

```
Input: [10, 1, -4, 8, 7, 19, 3, 17, 12]
[1, -4, 8, 7, 3]                  *10* [19, 17, 12]
([-4] *1* [3, 7, 8])              *10* ([17, 12] *19* [])
(         ([] *3* [7, 8]))        *10* (([] *12* [17]) [19])
(         ([] *3* ([] *7* [8])))
```

Side note

- Merge sort and quick sort are similar and different in the same was as insertion sort and selection sort
- for merge sort, splitting is “easy”, but combining is complex
- for quick sort, splitting is more complex, but combining is trivial
- Fill in the details by implementing the different sorting algorithms
- These algorithms will be revisited