Part 20

—

Sorting and Selection

CSCI 3110 Code

Summer 2015

1 Introduction

Here we implement the different sorting algorithms discussed in class: Insertion Sort, Merge Sort, and the various variants of Quick Sort. We also discuss how to implement linear-time selection (because it's closely related to and used by Quick Sort) and how to construct a uniform random permutation of an input list in linear time, again because one of the Quick Sort variants uses this.

2 Insertion Sort

In an imperative language, Insertion Sort has appeal due to its simplicity. In Haskell, Merge Sort is ridiculously easy to implement, as we will see below, so there is little motivation for using Insertion Sort. However, we present it here anyway because we will use it as part of our linear-time selection algorithm:

```
insertionSort :: Ord a ⇒ [a] → [a]
insertionSort = insertionSortBy compare

insertionSortBy        :: (a → a → Ordering) → [a] → [a]
insertionSortBy cmp xs = insSort xs
  where insSort []       = []
        insSort (x : xs) = ins (insSort xs)
          where ins []             = [x]
                ins ys@(y : ys') | cmp x y ≡ GT = y : ins ys'
                                 | otherwise    = x : ys
```

So, to sort the empty list, we don't need to do anything. To sort a non-empty list $x : xs$, we first sort xs and then insert x into the result. Inserting x into a list ys produces the list $x : ys$ if x is no greater than the head of ys—the resulting list is sorted in this case. Otherwise, we recursively insert x into ys tail.

3 Merge Sort

The strategy of Merge Sort is straightforward in a world where we know the size of the input: recursively sort the left half, recursively sort the right half, then merge the two lists. If we don't know the input size (because we don't want to traverse the input list only to figure out this size), we don't know where the left half ends and the right half begins. However, we can do exactly what I told you not to do in class: consider how the recursion unfolds. At the bottom of the recursion, the merge process starts with trivially sorted singleton lists. As we work our way back up the recursion tree, we form bigger and bigger sorted lists by merging them in a pairwise fashion. This is easy enough to implement in Haskell:

```
mergeSort :: Ord a ⇒ [a] → [a]
mergeSort = mergeSortBy compare

mergeSortBy        :: (a → a → Ordering) → [a] → [a]
mergeSortBy cmp xs = mSort xs
  where mSort [] = []
        mSort xs = ms $ map (:[]) xs
          where ms [s]          = s
                ms ss           = ms (m2 ss)
                m2 (s1 : s2 : ss) = mergeBy cmp s1 s2 : m2 ss
                m2 ss           = ss
```

So *mergeSort* first turns every input element into a singleton list using *map* (:[ ]) and then calls *ms* on the resulting list of lists. If *ms* is given a single list, this is the final sorted list, so it returns it. Otherwise, it applies *m2* once to the current list of lists and then recurses on the result. *m2* merges the lists in the given list of lists in a pairwise fashion. If there are at least two lists, it takes the first two, merges them, and prepends the result to the list produced by calling *m2* recursively on the remaining lists. Otherwise, it just returns the current list of lists. *merge* itself is straightforward to implement, and I present it here without further discussion:

```
merge :: Ord a ⇒ [a] → [a] → [a]
merge = mergeBy compare

mergeBy         :: (a → a → Ordering) → [a] → [a] → [a]
mergeBy cmp ls rs = mrg ls rs
  where mrg ls        []         = ls
        mrg []        rs         = rs
        mrg ls@(l : ls') rs@(r : rs') | cmp l r ≡ GT = r : mrg ls  rs'
                                      | otherwise    = l : mrg ls' rs
```

4 Quick Sort

Quick Sort is also simple enough. We pick an element from the input as a pivot and partition the input list into three sublists, one containing only elements less than the pivot, one containing elements equal to the pivot, and one containing elements greater than the pivot. We recursively sort the first and last lists and concatenate the results:

```
quickSort                :: Ord a ⇒ ([a] → a) → [a] → [a]
quickSort findPivot xs = qs xs
  where qs [ ] = [ ]
        qs xs  = qs ls ⧺ ms ⧺ qs rs
          where (ls, ms, rs) = partition (findPivot xs) xs
```

This *quickSort* function takes a function *findPivot* as its first argument, which is used to select the pivot from the input. The pivot finding strategy is the main difference between different variants of Quick Sort. While this is intuitively the right definition (and would be perfectly fine in an imperative language), we have a bit of a problem: the type of *findPivot* isn't quite right. Since it's a pure function, it behaves the same every time we call it on a given input list. However, we want to be able to pick a random pivot in our *randomPivotQuickSort* implementation. To do this, we implement Quick Sort in the *State* monad. The state is the current pivot selector. The pivot selector itself can then replace itself with an updated pivot selector if necessary to ensure we use a pseudo-random sequence of pivot selectors

```
newtype PivotSelect a = PS {runPS :: [a] → State (PivotSelect a) a}

getPivot    :: [a] → State (PivotSelect a) a
getPivot xs = get ≫= flip runPS xs
```

quickSort now runs in this pivot monad:

```
quickSort              :: Ord a ⇒ PivotSelect a → [a] → [a]
quickSort findPivot = quickSortBy findPivot compare

quickSortBy                      :: PivotSelect a → (a → a → Ordering) → [a] → [a]
quickSortBy findPivot cmp xs = evalState (qs xs) findPivot
  where qs [ ] = return [ ]
        qs xs  = do p ← getPivot xs
                    let (ls, ms, rs) = partitionBy cmp p xs
                    sls ← qs ls
                    srs ← qs rs
                    return (sls ⧺ ms ⧺ srs)
```

Now, *partition* is straightforward to implement again, so I present it without comment:

```
partition :: Ord a ⇒ a → [a] → ([a], [a], [a])
partition = partitionBy compare
```

```
partitionBy          :: (a → a → Ordering) → a → [a] → ([a],[a],[a])
partitionBy cmp p = part
  where part [ ]       = ([ ],[ ],[ ])
        part (x : xs) | cmp x p ≡ LT = (x : ls,     ms,     rs)
                      | cmp x p ≡ EQ = (    ls, x : ms,     rs)
                      | otherwise    = (    ls,     ms, x : rs)
          where (ls, ms, rs) = part xs
```

As already said, the different implementations of *quickSort* differ mainly in how they choose the pivot. The simplest strategy is to simply choose the first input element as pivot:

```
simpleQuickSort :: Ord a ⇒ [a] → [a]
simpleQuickSort = simpleQuickSortBy compare

simpleQuickSortBy :: (a → a → Ordering) → [a] → [a]
simpleQuickSortBy = quickSortBy pickFirst

pickFirst :: PivotSelect a
pickFirst = PS (return ∘ head)
```

This is extremely simple and works very well if the input is a uniformly random permutation: the expected running time is $O(n \lg n)$ in this case because the pivot is very likely to split the input roughly in half most of the time. An almost sorted input, on the other hand, forces *simpleQuickSort* to take quadratic time. We can guarantee that the pivot splits the input in half by choosing the pivot to be the median of the input:

```
worstCaseQuickSort :: Ord a ⇒ [a] → [a]
worstCaseQuickSort = worstCaseQuickSortBy compare

worstCaseQuickSortBy       :: (a → a → Ordering) → [a] → [a]
worstCaseQuickSortBy cmp = quickSortBy (pickMedianBy cmp) cmp

pickMedianBy       ::  (a → a → Ordering) → PivotSelect a
pickMedianBy cmp =  PS $ λxs → return (worstCaseSelectBy cmp (|xs| div 2) xs)
```

worstCaseSelect k xs selects the *k*th smallest element in *xs* using linear-time selection and is discussed below. This guarantees that the running time is in $O(n \lg n)$, but *worstCaseSelect* is not a cheap operation, so this is more of an academic exercise. What we would like is a simple algorithm that exhibits the same expected running time as *simpleQuickSort* but is independent of the input permutation. This would guarantee that an adversary cannot force the algorithm to take quadratic time by providing a particularly bad input permutation. The first strategy is to *guarantee* that the input is a uniform random permutation, at which point we can safely use *simpleQuickSort*:

```
randomPermutationQuickSort :: (Ord a, RandomGen g) ⇒ g → [a] → [a]
randomPermutationQuickSort = randomPermutationQuickSortBy compare
```

```
randomPermutationQuickSortBy           :: RandomGen g ⇒ (a → a → Ordering) → g → [a] → [a]
randomPermutationQuickSortBy cmp g = simpleQuickSortBy cmp ∘ randomPermutation g
```

randomPermutation is a linear-time function that uses a random number generator *g*, which must be provided as part of the input.[1] *randomPermutation* is provided by *Algos*.*Random*.*Permuting*.

There is a second strategy to avoid the worst case using randomness. To come up with it, we need to look a little closer at why *simpleQuickSort* does well on random inputs. The reason is that every element is equally likely to be the first element in the input and thus is equally likely to be chosen as the pivot if the input is a random permutation. This suggests choose the pivot uniformly at random from the input list as an alternate strategy for randomized Quicksort. This is the strategy used by *randomPivotQuickSort*:

```
randomPivotQuickSort :: (Ord a, RandomGen g) ⇒ g → [a] → [a]
randomPivotQuickSort = randomPivotQuickSortBy compare

randomPivotQuickSortBy           :: RandomGen g ⇒ (a → a → Ordering) → g → [a] → [a]
randomPivotQuickSortBy cmp g = quickSortBy (pickRandom g) cmp

pickRandom    :: RandomGen g ⇒ g → PivotSelect a
pickRandom g = PS $ λxs → do let (i, g′) = randomR (0, |xs| − 1) g
                             put (pickRandom g′)
                             return (xs !! i)
```

5 Selection

As discussed in class, we can implement linear-time selection using a minor adaptation of *quickSort*. Again, we have a choice in how we choose the pivot, so we obtain:

```
select              :: Ord a ⇒ PivotSelect a → Int → [a] → a
select findPivot = selectBy findPivot compare
```

[1]Getting a hold of a good random number generator in Haskell is is notoriously hard; since it's inherently impure, we can really only do this in the *IO* monad.

```
selectBy                          :: PivotSelect a → (a → a → Ordering) → Int → [a] → a
selectBy findPivot cmp k xs = evalState (sel k xs) findPivot
  where sel k xs = do p ← getPivot xs
                      let (ls, ms, rs) = partitionBy cmp p xs
                          nls          = |ls|
                          nms          = |ms|
                      if k < nls
                        then sel k ls
                        else if k < nls + nms
                               then return (head ms)
                               else sel (k − nls − nms) rs
```

The simple deterministic and the two randomized versions of *select* are analogous to their counterpart variants of *quickSort*:

```
simpleSelect :: Ord a ⇒ Int → [a] → a
simpleSelect = simpleSelectBy compare

simpleSelectBy :: (a → a → Ordering) → Int → [a] → a
simpleSelectBy = selectBy pickFirst

randomPermutationSelect :: (Ord a, RandomGen g) ⇒ g → Int → [a] → a
randomPermutationSelect = randomPermutationSelectBy compare

randomPermutationSelectBy          :: RandomGen g ⇒ (a → a → Ordering) → g → Int → [a] → a
randomPermutationSelectBy cmp g k = simpleSelectBy cmp k ∘ randomPermutation g

randomPivotSelect :: (Ord a, RandomGen g) ⇒ g → Int → [a] → a
randomPivotSelect = randomPivotSelectBy compare

randomPivotSelectBy          :: RandomGen g ⇒ (a → a → Ordering) → g → Int → [a] → a
randomPivotSelectBy cmp g = selectBy (pickRandom g) cmp
```

worstCaseQuickSort punted by calling *worstCaseSelect* to find the median of the input elements. Clearly, *worstCaseSelect* cannot do this because finding the median is just selection. So we employ the strategy from class: Instead of using the median as the pivot, we find the median of a 5-sample of the input, which is guaranteed to split the input 30-70 or better:

```
worstCaseSelect :: Ord a ⇒ Int → [a] → a
worstCaseSelect = worstCaseSelectBy compare

worstCaseSelectBy        :: (a → a → Ordering) → Int → [a] → a
worstCaseSelectBy cmp = selectBy (pickApproxMedianBy cmp) cmp
```

```
pickApproxMedianBy     :: (a → a → Ordering) → PivotSelect a
pickApproxMedianBy cmp = PS $ λxs → return (go xs)
  where go [x] = x
        go xs  = worstCaseSelectBy cmp (⌊|ys| div 2) ys
          where ys = fiveSampleBy cmp xs

fiveSampleBy     :: (a → a → Ordering) → [a] → [a]
fiveSampleBy cmp = unfoldr pickSample
  where pickSample [ ] = Nothing
        pickSample xs  = Just (m, zs)
          where (ys, zs) = splitAt 5 xs
                m        = sortSelectBy cmp (⌊|ys| div 2) ys
```

The *sortSelect* function just uses the naïve strategy of sorting the input using *insertionSort* and then picking the *k*th element:

```
sortSelectBy          :: (a → a → Ordering) → Int → [a] → a
sortSelectBy cmp k xs = insertionSortBy cmp xs !! k
```

6 Sorting in Expected Linear Time

If the input elements are real numbers sampled uniformly at random from a certain range, then we can sort them in expected linear time:

```
bucketSort :: Real t ⇒ [t] → [t]
bucketSort = bucketSortBy realToFrac

bucketSortBy        :: (a → Double) → [a] → [a]
bucketSortBy key xs = concatMap (insertionSortBy $ comparing key) (elems a)
  where n          = |xs|
        keys       = map key xs
        minKey     = minimum keys
        maxKey     = maximum keys
        scale      = fromIntegral n / (maxKey − minKey)
        buckets    = map (λk → ⌊(k − minKey) · scale⌋) keys
        bucketvals = zip buckets xs
        a          = fmap reverse $ accumArray (flip (:)) [ ] (0, n) bucketvals
```

We determine the range of key values and divide it into n equal intervals. Then we map every element in the input to a bucket corresponding to one of these intervals and collect the elements in each bucket. We sort each bucket and concatenate the results. We proved in class that this takes expected linear time if the input keys are uniformly distributed.

7 Sorting Integers in Linear Time

The final two sorting algorithms here are ones we do not discuss in class, but they are tremendously useful. The first one, *countingSort*, sorts n integers between 1 and n in $O(n)$ time.[2] The second one, *radixSort* is generally described in terms of boosting the range of integers that can be sorted in linear time to polynomial. Specifically, integers between 1 and n^k can be sorted in $O(kn)$ time using *radixSort*. We will be mostly interested in using *radixSort* to sort edges over n vertices efficiently. Thus, we generalize it a bit so it can sort any kind of items by any sequence of k predicates that are integers between 1 and n. If this is confusing right now, don't worry, we'll get to it. First we'll look at *countingSort*. Similar to *radixSort*, we'll make this one a bit more general, too:

```
countingSortBy :: (a → Int) → [a] → [a]
```

countingSortBy key xs produces a list xs' containing the same elements as *xs* but sorted by their *key*s. That is, if x predeces y in xs', then $\mathit{key}\ x \leqslant \mathit{key}\ y$. If $\mathit{minKey} = \mathit{minimum}\ (\mathit{map}\ \mathit{key}\ \mathit{xs})$ and $\mathit{maxKey} = \mathit{maximum}\ (\mathit{map}\ \mathit{key}\ \mathit{xs})$, then *countingSortBy* takes $O(1 + \mathit{xs} + \mathit{maxKey} - \mathit{minKey})$ time. Thus, the following version of *countingSortBy* specialized to integers takes $O(n)$ time to sort n integers between 1 and n.

```
countingSort :: Integral t ⇒ [t] → [t]
countingSort = countingSortBy fromIntegral
```

It remains to implement *countingSortBy*:

```
countingSortBy _   [] = []
countingSortBy key xs = concat $ elems a
  where keys    = map key  xs
        keyvals = zip  keys xs
        minKey  = minimum  keys
        maxKey  = maximum keys
        a       = fmap reverse $ accumArray (flip (:)) [] (minKey, maxKey) keyvals
```

So we first create a list of key-value pairs *keyvals* from the input list *xs* and determine the minimum and maximum keys, *minKey* and *maxKey*. Then we create an array of lists, each initially empty, and scan *keyvals* to prepend every value in this list to the list corresponding to its key. If all we cared about is obtaining the input elements in sorted order, we could now concatenate the lists of elements in increasing order of keys. That's in fact what we do using *concat* \$ *elems a*, but before doing this, we reverse every list the array using *fmap reverse*. We do this because we want *countingSortBy* to be stable, that is, if $\mathit{key}\ x \equiv \mathit{key}\ y$ and x precedes y in the input, then x should precede y in the output; elements should change their order only if absolutely necessary to ensure the output is sorted. Without *fmap reverse*, the output would be "anti-stable" (not a real term): if $\mathit{key}\ x \equiv \mathit{key}\ y$ and x precedes y in the input, then y is prepended to the $(\mathit{key}\ x)$th list in a after x and thus would precede x in the output.

[2]The functional implementation we use here does not pull some of the fancy tricks the imperative version in the textbook applies, simply because it doesn't have to.

Why do we care about stability? It's a useful property to have for a sorting algorithm and it is absolutely crucial for the correctness of our next algorithm, *radixSort*.

```
radixSortBy :: [a → Int] → [a] → [a]
```

Now we are given not one key function but a whole sequence of them. If the functions we are given are $[key_1, key_2, \ldots, key_k]$, then we want the output to be sorted primarily by key_k, secondarily by key_{k-1}, and so on down to key_1. Using this, we can easily sort n integers between 1 and n^k:

```
radixSort     :: Integral t ⇒ [t] → [t]
radixSort [ ]  = [ ]
radixSort [x] = [x]
radixSort xs   = radixSortBy digits xs
  where n           = |xs|
        maxx        = maximum xs
        k           = 1 + ilog n (fromIntegral maxx)
        digits      = map digit (take k divs)
        divs        = 1 : map (·n) divs
        digit dv x = (fromIntegral x div dv) mod n

        ilog base x = go 0 x
          where go lg y | y < base   = lg
                        | otherwise = go (lg + 1) (y div n)
```

radixSort itself doesn't do anything interesting beyond figuring out the *key* functions for extracting the digits, to be passed to *radixSortBy*. So we first find the number n of elements in *xs* and the maximum element *maxx* in *xs*. The number of digits by which to sort is the number of digits needed to represent *maxx* in base-n. That's what $1 + ilog\ n\ maxx$ computes. *divs* is the sequence of divisors of the digits from lowest to highest, $[1, n, n^2, n^3, \ldots]$. Finally, we construct a sequence of *key* functions, *digits*, by mapping *digit* over the first k values in this list of divisors. For a given divisor *dv*, *digit dv x* simply computes x's digit value by dividing by *dv* and taking the remainder modulo n of the result. By passing these digit functions to *radixSortBy*, we ensure the numbers are sorted primarily by their highest digit, secondarily by their second-highest digit, and so on, which is obviously the correct sorting order. Now on to implementing *radixSortBy*:

```
radixSortBy keys xs = foldl′ (flip countingSortBy) xs keys
```

Woah, that's it? We repeatedly sort the input, first by the first key in the list, then by the second, then by the third, and so on. Due to the stability of *countingSortBy*, this does the right thing: The final sort arranges the elements so that the elements are sorted by their final key. All elements with the same final key are kept in the same order as in the input to the final sort. By induction, the input to the final sort was sorted by the first $k - 1$ keys. Thus, the final output is sorted primarily by the final key, secondarily by the second-last key, and so on.