Set 4 Homework, Analysis of Algorithms

Jay R Bolton

May 27, 2012

- p 194: 8.1-4, p 197: 8.2-4, p 200: 8.3-2, p 204: 8.4-2, p 206: 8.3 or 8.4
- p 215: 9.1-1, p 219: 9.2-1, p 223: 9.3-8 , 9.3-9, p 224: 9-2

Chapter 8

8.1-4 To sort each k sublist, we will use an efficient comparison sort ($\Omega(n\ lg\ n)$).

$$\begin{aligned}
&T(n) = k\Omega(n/k\ lg\ n/k) + \Theta(1)\\
&= kc(n/k\ lg\ n/k) + d\\
&= cn\ lg\ n/k + d\\
&\geq cn\ lg\ n/k\\
&\geq cn\ lg\ k \qquad \text{because k is a constant}
\end{aligned}$$

Really not sure if I did that right.

8.2-4 First do counting sort up to line 9 ($\mathcal{O}(n+k)$) to get C. Then we get our output with:

$$\begin{aligned}
&result := C[a + (a - b)]\\
&result := result - C[a - 1] \quad if\ (a - 1) \geq 0
\end{aligned}$$

Which is $\Theta(1)$. That was an interesting/challenging puzzle.

8.3-2 Insertion sort is stable: if we scan left-to-right, we will insert elements their rightmost insertion slot, meaning that leftmost equal elements remain leftmost and rightmost remain rightmost.

Merge sort is also stable. The merge step scans left to right, and will insert the left element first between two equal elements.

Heap sort is not a stable sort. If you are performing BuildMaxHeap, and your current node is equal to the first left child, and you swap the current node with the parent node, then the child will move to the current position, thus reversing the order of the two equal nodes.

Quicksort is also not stable because of the partition function, which can swap elements out of order.

You can make any sort stable by pairing every element with its unique index. After the sort you can sort the indices of equal elements, for a worst case of also $n\ lg\ n$, which just adds a constant factor.

8.4-2 The worst case occurs when all n the elements go into a single buckeet.

If we simply used n buckets, it basically becomes counting sort and avoids the quadratic worst case.

8.3 **(a)** We cannot use counting or radix sort because we're not assuming they are all d digits or they are all in range 0 to k.

First, partition the list into buckets based on the number of digits. The worst case for this is $\mathcal{O}(n)$ with n being the number of digits. Worst case is when all elements are single digits.

We then have a series of sublists, each with equal numbers of digits, which satisfies the requirement for radix sort. Sort all those in

$$\mathcal{O}(m + k)$$

time, which will be faster than the first step.

(b) Here our ordering is a bit different. More characters does not necessarily mean higher value.

My only idea is to perform radix sort, sorting right-to-left. When we run out of leading characters, we simply stay on the leftmost one. So this will take $\mathcal{O}(m+n) = \mathcal{O}(n)$. Radix sort takes $\mathcal{O}(d(n+k))$ to sort, where k is a constant. Our worst case for d is n, the total number of digits, in which case n would be 1, so $\mathcal{O}(d)$. n will always be less than or equal to d.

Chapter 9

9.1-1
```
  if length list < 2 return nil
  if list[1] < list[2] then min = list[1] and snd = list[2]
  else min = list[2] and snd = list[1]
for i = 3..(length list)
    if list[i] < min then min = list[i]
    else if list[i] < snd then snd = list[i]
  return snd
```

The complexity of each individual line is $\Theta(1)$, and the loop goes through $\Theta(n - 2)$ times, so our complexity is $\Theta(n)$. Not sure where his additional $lg\ n$ comes from.

9.2-1 p is the beginning index of our list and r the end. The very first line of the function states that if $p == r$, or if the list has zero elements, then we return.

If the left list of RandomizedPartion is empty, then k is going to be 1 and either i is equal to it, then we return non-recursively, or greater, then we recurse on the right list (if less, then the algorithm is incorrect). The only case in which we would recurse on an empty list is if i is given out of bounds.

9.3-8 This problem is equivalent to finding the $2n/2 = n^{th}$ smallest element among the two lists.

My first instinct was to do a merge from merge sort, but that will take n comparisons.

Instead we could:

(a) Index the middle element of both lists. If they are the same, return that element

(b) If one is greater and one less, then repeat step 1 on the following sublists:

   i. The list from index 0 to our larger median

   ii. The list from our smaller median to n

We can repeat this until we find equal medians or until we only have two or three elements left.

9.3-9 You'd want the main pipeline to have an equal average distance to all the wells.

Using just the y-coordinates (the x-coordinate is irrelevant since the main line is going east-west), take the mean, which is a subcase of the selection problem and has complexity $\mathcal{O}(n)$

9-2 **(a)** We're simply giving everything the same weight whose size is evenly distributed over the total number of elements. This will just cause us to choose the middle element.

(b) First, sort the list of elements (not the weights) in $\mathcal{O}(n\ lg\ n)$ time. Then simply iterate on the list until the sum of the weights of the elements becomes greater than 1/2, at which point choose the previous element.

(c) As in Select, insertion-sort $n/5$ groups of sublists. Then select the median weight from each. Find the median weight of those median weights. Sum the weights from the top-left quadrant (all less than x). If it is greater than x including x, then stop. Otherwise continue recursion with bottom-right quadrant.