COMP 3711H – Fall 2016
Tutorial 2 – Solution Sketch

Note. The answers given below might not be complete. Some are sketches or hints, sometimes missing details. Also answers to somne of thee simplest questions might not be given.

1. There are n items in an array. It is easy to see that that their minimum can be found using $n-1$ comparisons and that $n-1$ are actually required. It is also easy to see that finding the max can similarly be done using $n-1$ comparisons with $n-1$ required.

   Design an algorithm that finds *both* the minimum and the maximum using at most $\frac{3}{2}n+c$ comparisons where $c>0$ can be any constant you want.

   Note: Although it is harder to prove, $\frac{3}{2}n+c$ comparisons is actually a lower bound.

   Assume n is even. Let the items be $x_0, x_1, \ldots, x_{n-1}$. Divide the items into $n/2$ pairs, $p_i = \{x_{2i}, x_{2i+1}\}$. Using $n/2$ comparisons compare the items in each p_i to each other, finding $m_i = \min\{x_{2i}, x_{2i+1}\}$ and $M_i = \max\{x_{2i}, x_{2i+1}\}$. Using $n/2-1$ comparisons find the minimum of the m_i and using another $n/2-1$ comparisons find the maximum of the M_i. These are respectively, the min and max of the entire set and have been found using $3n/2-2$ comparisons. If n is odd, keep one item x on the side and perform the algorithm above. Then use two comparisons to compare x to the min and max found to compute the min and max of the entire set. This will use $3(n-1)/2-2+2 = 3n/2-3/2$ comparisons.

2. Prove that insertion in a binary search tree requires at least $O(\log n)$ comparisons (in the worst case) per insertion, where n is the number of items in the search tree.

   Hint: What lower bounds have we learned in class? Suppose you built the search tree using insertions. What can you do with it?

   After inserting into a binary search tree it takes only O(n) time to read the items out using an INORDER scan. This would produce the items in sorted order. This means that if you could insert in o(n log n) time (time being number of comparisons) you could sort in o(n log n) time, which we proved in class was not possible.

3. Build a Binary Search Tree for the items
   8, 4, 6, 13, 3, 9, 11, 2, 1, 12, 10, 5, 7
   and draw the final tree.
   Now, delete 3, 9, 4 in order and draw the resulting trees.

4. The maximum item in a set of n real-valued keys is well defined. The maximum item in a set of n 2-dimensional real-valued points is not.

   One definition that is used in database theory is that of *skyline vectors*. These are also known as *maximal points* or *maximal vectors*.

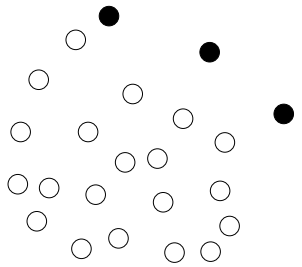
   Let $S = \{p_1, p_2, \ldots, p_n\}$ be a set of 2-d points where $p_i = (x_i, y_i)$. A point $p \in S$ is a *skyline vector* if no other point is bigger than it in both x and y dimensions.

   Formally p_j *dominates* p_i if

   $$x_i < x_j \quad \text{and} \quad y_i < y_j.$$

$p = (x, y)$ *is a skyline vector in* S if no p_i in S dominates p.

In the example below, the 3 filled points are the skyline ones.



(a) Give an algorithm that finds the skyline vectors in a set S of n points in $O(n \log n)$ time.

(b) Suppose that the points all have integer coordinates in the range $[1, \ldots, n^2]$. Give an $O(n)$ algorithm for solving the same problem.

To simplify we will ssume that all the points have different x *and* y *coordinates (repeated values require just a few more comparisons.).*

(a) Sort the items from smallest to largest x *coordinate in* $O(n \log n)$ *time and insert the items into an array in sorted order.*

So the points are now $p_i = (x_i, y_i)$ *with*

$$x_1 \leq x_2 \leq \cdots \leq x_n.$$

Note that with this ordering, p_i *is maximal if and only if*

$$y_i > \max_{j>i} y_j$$

This gives a simple algorithm.

i) Report that p_n *is maximal. Set* $maxy = y_n$.

ii) Walk from right to left in the array, i.e., looking at p_i *with* $i = n - 1$ *to 1.*
If $y_i > maxy$, *report that* p_i *is maximal and set* $maxy = y_i$.

Total running time is $O(nlogn)$ *for the sort* $+O(n)$ *for the scan through the array.*

(b) replace the first sort by an $O(n)$ *time radix sort (using base* n *for the numbers).*