Comp 610 Lecture 2: Selection and Priority Queues Continued

We saw a new algorithm to find the median which was expected to use $3n/2 + o(n)$ comparisons.

It first selected a random subset S of $n/\log^2 n$ elements. It then sorted this set using $O(|S| \log |S|) = o(n)$ comparisons and found the $((k-n^{3/4})/\log^2 n)$th element l and the $((k+n^{3/4})/\log^2 n)$th element u of S. By comparing each element to a randomly chosen element of {u,l}, and to both if necessary, it split the elements other than u and l into the set L of those smaller than l, the set H of those bigger than u and the set M of those lying between the two pivots.

If $|L|<k$ and $|M|< n/\log^2 n$ and $|H|<n-k$ then it sorted M in o(n) comparisions and returned the $(k-|L|)^{th}$ element of M. Otherwise it sorted the entire set using n log n comparisons and returned the kth element. We saw that the expected number of comparisons used by the algorithm was $3n/2+o(n)$, unless it actually resorted to sorting all n elements. We claimed and will see, that the probability this occurred was $o(n^{-2})$. Can you see why this is true? This implies that the algorithm is expected to make $3n/2 + o(n)$ comparisons in total.

We saw that we could reduce the worst case number of comparisons in the trickle down operation from the root of a (subheap of a) heap as follows.

We find the "trickledown path" whose first element is the smallest child of the root, by repeatedly appending the smallest child of the current last element of the path until this element is a leaf. We noted that this path consists of a sequence $y_1,y_2,\ldots,y_j$ with $y_1 \leq y_2 \leq \ldots \leq y_j$. We can now insert the element x at the root into this sorted list and update the heap accordingly. We can find the position of x in the list using the round up of $\log_2 j$ comparisons.

In the first assignment, you will be asked to determine the number of comparisons this procedure uses to build a heap in the worst case. We mentioned that the expected time taken by a variant of this algorithm which actually looks for x's position starting at the end of the path (so essentially moving every element of the path up a level and then bubbling the element which was at the root up from the leaf position which contained the last element of the path) was about 1.52 comparisons.

We also saw a partial order know as a Binomial Tree (I called this Binomial Queue in the lecture but will use tree to be consistent with the text see pages 527 and 528 for a definition).

We noted that we could transform a binomial tree with $n=2^j$ elements into a heap, as follows. We have found the minimum which is the root of the binomial tree and

will also be the root of the heap. The root of the binomial tree has j children, with one of these children being the root of a binomial tree of size 2^i for each i between 0 and j-1.

If j is 1 or 2 then there is no work to do. If j=3 then we used the binomial tree of size 4 rooted at child of the minimum as the heap rooted at the left child of the minimum. We then compared the roots of the binomial trees of size 1 or 2, to find the minimum of the three elements they contain and made this the right child of the root. The remaining two elements were this element's children.

If j was at least 4, we began our transformation process by transforming the binomial tree of height 2^{j-1} rooted at a child of the root of the binomial tree into a heap of size 2^{j-1}. We then removed the deepest leaf l from this heap to make it a heap of size 2^{j-1}-1. This will be the subheap rooted at the right child of the root of the heap. We now took l and the remaining binomial trees rooted at children of the root and with j-1 comparisons constructed a binomial tree of size 2^{j-1} out of them. We then transformed this into a heap, which was the subheap rooted at the left child of the root of the heap.

Letting $T(2^j)$ be the number of comparisons this procedure used to transform a Binomial tree of size 2^j into a heap of size 2^j we have $T(2)=T(4)=0$, $T(8)=1$ and for j at least 4, $T(2^j)=2T(2^{j-1})+j-1=5(2^j)/8-j-1$.

Now, we can build a heap of size 2^j-1 by adding a dummy element which will lose to everybody, building a heap of size 2^j, removing the dummy element which will be a leaf and placing the element which was the only leaf at the lowest level in the position that the dummy node was in, and using at most j comparisons to bubble it up and restore the heap order. So we can build a heap with $n=2^j$-1 elements using at most 13n/8 comparisons. We saw that we could build a heap on n elements by building log n such perfect heaps. So the total number of comparisons w need is at most n(13/8+o(1)).

This algorithm is due to Gonnet and Munro.

We discussd comparison trees and information theory lower bounds, which was the logarithm of the number of leaves a tree for solving a giving problem must have. This yields a log n!= (1+o(1))nlog n lower bound on sorting. For building a heap on n elemnts it yields a lower bound of the logarithm of the product of the sizes of the subtres constructed which is about 1.36n. For min, max and median, it yields a bound of log n, which is not very good.

We discussed adversarial lower bounds. Here an adversary must answer your queries in a way which is consistent with his previous answers. When you give an output he is free to choose any input consistent with his answers as the actual input, so your output must be consistent with all such inputs. Thus finding the

minimum requires n-1 comparisons as if there are two nodes which have not yet been the larger in a comparison then either could be the minimum.

We obtained a (3n-3)/2 lower bound on finding the median of an odd number n of elements as follows. :

We have two sets L and H each of which is initially empty. Whenever we compare an element of H with an element of L, the element of H wins.
For the first n-1/2 comparisons involving an element which has not yet been put in either L or H, before doing the comparison we add an element to H and an element to L, using elements involved in the comparison as much as possible. We can and do also insist that this comparison becomes one between an element of H and an element of L.

We know we must have at least n-1/2 such comparisons as to find the median we must compare every element. So, after the last of these n-1/2 comparisons w have one element p which is neither in H nor L.

In any comparison involving p and an element in L, p is found to be larger..
In any comparison involving p and an element in H, p is found to be smaller.

When the algorithm is done, one possible order is L in any order consistent with answers,p,H in any order consistent with answers. Thus, p may be the median and must be output. If there is an x in L not having lost to an element of p+L, then the order could be L-x in any order consistent with the answers,p,x,H in any order consistent with the answers. In this case x would be the median, and this is impossible. Symmetrically every element of H must win a comparision with an element of H+p or we would be done. So there are at least
n-1/2 comparisions within H+p and n-1/2 within L+p, and hence 3n-3/2 comparisons in total.