Solutions for CS 521 Homework #2

October 21, 2011

Problem 1

We will use the fact that a heap on n nodes has exactly $\lceil n/2 \rceil$ leaves. This can be shown by induction; if a heap on $n-2$ nodes has exactly $\lceil (n-2)/2 \rceil$ leaves, then after we add two leaves, the number of leaves increases by one (one old leaf now has a child): $\lceil (n-2)/2 \rceil + 1 = \lceil n/2 \rceil$. The base cases are $n = 0$ and $n = 1$.

Let us label the nodes of the heap with their heights. The nodes at height 0 are the leaves. When we remove these from the heap, we have a smaller heap, whose leaves are precisely the nodes labeled 1. If we again remove all the leaves, the new set of leaves will be those nodes labeled 2, etc., etc.

Since an n-node heap has exactly $\lceil n/2 \rceil$ leaves, it has exactly $n - \lceil n/2 \rceil = \lfloor n/2 \rfloor$ branch nodes. Let us carry out the leaf-removing process described above, calling the heaps created H_0 (the initial one), $H_1, H_2, \ldots, H_k$. Then H_0 has n nodes, H_1 has $\lfloor n/2 \rfloor$ nodes, H_2 has $\lfloor \lfloor n/2 \rfloor /2 \rfloor = \lfloor n/4 \rfloor$ nodes, and so on. In general, H_i will have exactly $\lfloor n/2^i \rfloor$ nodes. We know that exactly $\lceil \lfloor n/2^i \rfloor /2 \rceil$ of these will be leaves, and these will be precisely the nodes labeled i. The number of nodes of height i is therefore

$$\left\lceil \frac{\lfloor n/2^i \rfloor}{2} \right\rceil \leq \left\lceil \frac{n/2^i}{2} \right\rceil = \left\lceil \frac{n}{2^{i+1}} \right\rceil ,$$

q.e.d.

Problem 2

Our goal here is to show that heapsort will take $\Omega(n \log n)$ time on the specified inputs. We assume all the elements are distinct. Without loss of generality, we will assume that the elements in the array are the integers from 1 to n. The key to showing the lower bound on the running time is to prove that many (here "many" will mean $\Omega(n)$) elements in the heap must work their way up to the root one step at a time. This is not automatically true for *every* element, since we often replace the value at the root with the last leaf value in the heap; this latter value jumps ahead and did not have to go up one step at a time.

Recall that during heapsort, we exchange the root with the last value in the heap and shrink the heap size so that the last value no longer belongs to the heap.

Lemma 1 *Let H be a heap on n nodes, let L be the set of its leaf nodes, and let V be the set of values (initially) contained in L. During the execution of heapsort, whenever the heap structure is valid,*[1] *the only values that will ever populate the nodes in L will be values taken from V.*

Proof: When the value x replaces the root value, it may be pushed down, and other elements may be pushed up to take its place. The only way for a value to enter a node in L is to be pushed down; thus x is the only value that can do so. But $x \in V$. □

Note that the nodes in L are being deleted from the heap, so that there will be many values in V that do not appear in L. This is fine.

The problem is easier to answer for an array in decreasing order, since it already has a max-heap structure. We want to show that $\lfloor n/2 \rfloor$ values do take the long way to the root. Notice that the values at the leaf nodes (of which there are $\lceil n/2 \rceil$) are all less than the values at all the branch nodes (of which there are $\lfloor n/2 \rfloor$). By

[1] i.e., after we have pushed the root value to its proper place in the heap

Lemma 1, for the first $\lfloor n/2 \rfloor$ iterations of heapsort, the only values in L will be the numbers $1, 2, \ldots, \lceil n/2 \rceil$; call this set V. However, the first $\lfloor n/2 \rfloor$ elements to be removed are $\lceil n/2 \rceil + 1, \lceil n/2 \rceil + 2, \ldots, n$; call this set, the elements at the original branch nodes, W. Every element of W must be removed from the heap while some node in L still exists. Therefore, no element of W can ever jump up to the root; it must be swapped up to reach there one level at a time. What is the total cost of all these swaps? Since at least half the branch nodes are in level $\lfloor \log n \rfloor - 2$ or deeper in the tree, the total cost is at least $\lfloor n/4 \rfloor \cdot (\lfloor \log n \rfloor - 2) = \Omega(n \log n)$. Thus heapsort will take $\Omega(n \log n)$ time on a descending array.

For an array in ascending order, it is difficult to see what the resulting heap will look like for general n. Fortunately, it turns out that the lower bound of $n \log n$ holds for heapsort run on *any* heap! The proof resembles the proof given above, but is a little more involved. We will list the facts you need to use; try to prove them yourself. (It is good practice for the exam!) If you can't, we give the proofs at the end of this document.

Lemma 2 *For a heap on n nodes with n even, there is a bijective mapping (a one-to-one correspondence) between the branch nodes and the leaf nodes such that every leaf node i is mapped to one of its ancestor*[2] *branch nodes j. In other words, we can match each leaf node to one of its ancestors above it, such that each branch node is matched to exactly one leaf.*

Try proving this with induction.

When leaf node i is matched with branch node j, we know $A[i] < A[j]$ by the properties of a max-heap. Use Lemma 2 to show

Lemma 3 *Let H be a heap on n nodes with n even, and let m be the true median of the elements of H (i.e., the average of the lower median and the upper median. m might not be an integer). Then at least half the branch nodes of H have values that are greater than or equal to m.*

(This lemma generalizes our earlier observation that for an array in decreasing order, the branch values were all greater than the leaf values; when n is even, the previous statement is the same as saying that all the branch nodes are greater than the median of the entire heap. In the lemma we are now saying *half* the branch nodes rather than *all* of them.)

Using Lemma 1 and Lemma 3, we can now show what we wanted to.

Theorem 4 *Heapsort on any n-element heap H takes $\Omega(n \log n)$ time.*

Proof: First suppose n is even. We want to show that at least $\lfloor n/4 \rfloor$ values must take the long way to the root, as described earlier. Define B to be the initial set of the branch nodes of H, and define L to be the initial set of the leaf nodes of H. By Lemma 3, at least half the values of B (call these B') are greater than the median of H. These values must be removed from the heap while some node in L still exists, because by the time L is empty, every element greater than the median has already been removed. Because of this fact and Lemma 1, no element of B' can ever jump up to the root; it must be swapped up to reach there one level at a time. What is the total cost of all these swaps? The cost is lowest when the nodes in B' are as high up in the heap as possible. Suppose that is the case. The set B' consists of at least $\lfloor n/4 \rfloor$ nodes; at least half of them are in level $\lfloor \log n \rfloor - 3$ or deeper in the tree, so that the total cost is at least $\lfloor n/8 \rfloor \cdot (\lfloor \log n \rfloor - 3) = \Omega(n \log n)$. Thus heapsort will take $\Omega(n \log n)$ time on H.

If n is odd, run one iteration on heapsort to leave a heap H' on $n-1$ nodes. Since $n-1$ is even, our argument shows that the remainder of the heapsort on H' will take $\Omega((n-1)\log(n-1)) = \Omega(n \log n)$ time. □

Problem 3

Initialize an empty min-heap H and an empty output list L. We will augment each element in input list i with a tag i, so that we can tell from which list a given element came. Begin by placing the first element of each input list into H. Now remove the minimum from H and add it to L. This minimum came from some

[2] An *ancestor* of a node y is a node that lies on the path from y to the root.

input list i; take the next element of input list i and place it in H (if there is no next element, do nothing). Repeat this process until all the input lists have been consumed. L will contain all the elements in sorted order.

The size of H never grows beyond k, so each operation will run in $O(\log k)$ time. Since we perform $2n$ operations, the total running time is $O(n \log k)$.

Problem 4

Insertion sort is, in general, an excellent choice for sorting nearly sorted input. On an n-element array with k elements in the wrong order, insertion sort will run in $O(kn)$ time. On an n-element array where each element is at most s spaces away from its proper place, insertion sort will run in $O(sn)$ time. We can combine these conditions: suppose we have an n-element array where each element, except for k of them, is at most s spaces away from its proper place. Then insertion sort will run in $O((k+s)n)$ time. If $k+s = o(\log n)$, then insertion sort will beat out a sort that takes $\Theta(n \log n)$ time, like mergesort or heapsort. In this problem, it is reasonable to assume that k and s are constant or nearly constant: most of the merchants will cash checks quickly (thus s is small), and very few of the merchants will wait a very long time to cash them (thus k is small).

Recall that quicksort will run in $O(n \log n)$ time ***if*** we choose a suitable pivot each time — we can use the linear-time selection algorithm to find the median, or we can choose a pivot at random and have an $O(n \log n)$ expected running time. Quicksort, however, has no way to take advantage of the fact that the array is almost sorted! Even if the partition splits the array in half each time, it must still go through all the motions of the sort: the recursion tree will be $\log n$ levels deep, and the partitioning in each level of the tree will take $\Theta(n)$ work. The best-case running time for quicksort is therefore $\Theta(n \log n)$. For this problem, insertion sort will tend to do better.

Problem 5

Since regular quicksort runs in $O(n \log n)$ time, it is reasonable to assume that quicksort stopping at subarrays of size k will run in $O(n \log(n/k))$ time. If quicksort partitions at the median each time, the number of levels in the recursion tree will be approximately $(\log n)/k$, since we start with an array of n elements and halve it repeatedly until each size is approximately k. Even if the partition is not this good but still splits at a constant proportion each time, the runtime will be $O(n \log(n/k))$ (with a different constant depending on the quality of the split). The subsequent insertion sort will then have to move each element into its proper place. Notice that each element only has to be moved k spaces at most. The running time for this insertion sort will therefore be $O(nk)$. Adding the two runtimes gives us the desired answer.

Let us suppose our bound is tight, so that the worst-case running time is, in fact, $\Theta(nk + n \log(n/k))$. If $k = \Theta(\log n)$, the running time is $\Theta(nk + n \log(n/k)) = \Theta(n \log n + n \log n - n \log \log n) = \Theta(n \log n)$. If $k = \omega(\log n)$, the running time is $\omega(n \log n + n \log(n/k)) = \omega(n \log n)$; this is too slow. We should therefore choose $k \leq c \cdot \log n$ for some constant c. Since small values of k will not have much of an effect, a reasonable guess is that the optimal result will happen for $k \approx c \cdot \log n$ for some constant c.

How should we choose c? The theory does not have an answer for this. It is likely to depend on the constant hidden by our asymptotic notations. The best way to determine it is by experimentation. There are many complications: for example, on modern processors, the speed of a program depends heavily on whether it can remain in cache memory most of the time or needs to reach out to main memory frequently. The optimal value of k is likely to depend on this cache behavior.

Problem 6

If we divide into groups of 7, the algorithm will still run in linear time. If we divide into groups of 3, it will take $\Theta(n \log n)$ time!

Let us form the recurrence $T(n)$ for the running time when we divide into groups of 7. Our first step is to sort each group; this will be linear time altogether. Next, we run the algorithm on the medians of

the groups; this will take $T(\lfloor n/7 \rfloor)$ time. Finally, we re-run the algorithm on the elements that have not been eliminated; there are approximately $5n/7$ left. We will bound this from above as $\lfloor 5n/7 \rfloor + r$ for some constant r. Our recurrence is

$$T(n) = T(\left\lfloor \frac{n}{7} \right\rfloor) + T(\left\lfloor \frac{5n}{7} \right\rfloor + r) + n.$$

An easy way to tell that $T(n)$ is linear is that the sum of the fractions in the recursive calls $(1/7 + 5/7)$ is less than one. A rigorous proof is not too difficult. Let us try to show $T(n) \leq cn$ by induction. Assume $T(m) \leq cm$ for all $m < n$.

$$\begin{aligned} T(n) &= T(\left\lfloor \frac{n}{7} \right\rfloor) + T(\left\lfloor \frac{5n}{7} \right\rfloor + r) + n \\ &\leq c(\left\lfloor \frac{n}{7} \right\rfloor) + c(\left\lfloor \frac{5n}{7} \right\rfloor + r) + n \\ &\leq c(\frac{n}{7}) + c(\frac{5n}{7} + r) + n \\ &= c(\frac{6n}{7} + r) + n. \end{aligned}$$

We need this to be $\leq cn$:

$$\begin{aligned} c(\frac{6n}{7} + r) + n &\leq cn \\ n &\leq c(\frac{n}{7} - r) \\ \frac{n}{n/7 - r} &\leq c. \end{aligned}$$

The left-hand side approaches 7 for large n. It is clear that for some n_0, some value of c will make this true for all $n > n_0$.

The story is different for groups of 3. Here our recurrence will be

$$T(n) = T(\left\lfloor \frac{n}{3} \right\rfloor) + T(\left\lfloor \frac{2n}{3} \right\rfloor + r) + n,$$

since the recursive call is made on $\lfloor n/3 \rfloor$ medians and approximately $2/3$ of all the elements must be carried over to the next step. Here the fractions add to 1, so the recurrence does not behave linearly. Let us try to solve as we did above:

$$\begin{aligned} T(n) &= T(\left\lfloor \frac{n}{3} \right\rfloor) + T(\left\lfloor \frac{2n}{3} \right\rfloor + r) + n \\ &\leq c(\left\lfloor \frac{n}{3} \right\rfloor) + c(\left\lfloor \frac{2n}{3} \right\rfloor + r) + n \\ &\leq c(\frac{n}{3}) + c(\frac{2n}{3} + r) + n \\ &= c(n + r) + n = n(c + 1) + cr. \end{aligned}$$

We need this to be $\leq cn$, but this is clearly impossible. Changing the value of r will not help; even if we make it negative, $n(c + 1)$ still grows too fast, and cn cannot catch up to it. The solution to this recurrence is in fact $\Theta(n \log n)$. See our solution to Problem 4d on Homework #1 for the proof of a similar recurrence.

Problem 7

We can find the median m and then find the k elements of A whose difference from m is the smallest in absolute value. Let the input be given in an array A. Find the median of A using the linear-time algorithm; let us call the median m. Build an array $B[1..n]$, assigning $B[i] = |A[i] - m|$. Now we find the k-th order statistic of B (again in linear time) and partition around it, but every time we perform a swap in B, we perform the same swap in A. At the end, $B[1..k]$ will contain the k smallest elements of B. The corresponding elements in A will be the k input numbers that are closest in value to m. Therefore we return $A[1..k]$.

Problem 8

We find the k-th quantiles by repeated applications of the order statistic algorithm and partitioning. The key to achieving an $O(n \log k)$ running time is to split the work evenly (or almost evenly) at each level.

The method is clearest when k is a power of 2 and n is one less than a multiple of k; this latter case ensures that every subarray we consider has a true median and we do not have to choose upper or lower. Recall that there is some constant $C > 0$ such that the order-statistic algorithm and the partition run in Cn time. Our first step is to find the median and partition around it. We have found the 2nd quantile (i.e., the median) in Cn time. To find the 4th quantiles, we find the median of each half of the array and partition each half around its median. We have found the 4th quantiles in an extra $2 \cdot C(n/2) = Cn$ time. To find the 8th quantiles, we find the median of each fourth of the array and partition each fourth around its median. We have found the 8th quantiles in an extra $4 \cdot C(n/4) = Cn$ time. We can continue like this; for $k = 2^i$, we will need i iterations, but each iteration only takes Cn time. (Notice that we only partition as large an array as we need to at each step.) The total running time is $\leq Cni = Cn \log k = O(n \log k)$.

We now give our algorithm for general k and n; we assume $n \geq k \geq 2$. Let k' be the smallest power of two not less than k. Notice $k' < 2k$. Perform the process described above using k' instead of k; when we have to choose the median of an even-length subarray, we can choose either the upper median or lower median. Record the location of each median found in an array M; when we are done, sort M in $O(k' \log k') = O(k \log k)$ time.

To find the k-th quantiles, we first create a list L of the order statistics we will need (see Section 8.1 below). For each order statistic s, we go to location s in the array and look to see between which two elements of M it lies. (Since M is sorted, we can use binary search; all these searches will therefore run in $O(k \log k)$ time.) If location s is in fact present in M, the s-th order statistic is just $A[s]$. Otherwise, s lies between two adjacent elements of M, say $M[i]$ and $M[j]$, where $j = i + 1$.[3] Run the order-statistic and partitioning algorithm on $A[M[i] + 1..M[j] - 1]$ to find the s-th order statistic. Since we divided A into k' approximately equal sections, each subarray $A[M[i] + 1..M[j] - 1]$ has size at most n/k'. The total cost of the partitioning will therefore be $C(n/k') \cdot k < 2Cn$.

Let's add up all the running times: the partitioning into k' subarrays took $O(n \log k') = O(n \log k)$ time; sorting the median array M took $O(k \log k)$ time; finding the position of the order statistics in M took $O(k \log k)$ time; and the partitions using these order statistics took $O(n)$ time. The sum of these four running times is $O(n \log k)$.

(We can also solve the problem more directly without first partitioning into k' subarrays; when k is even we find the median, and when k is odd we find one of the two k-th quantiles closest to the median. Either way, the array is split approximately in half, and we can recurse on each part. The trouble with this method is that it is tricky to ensure that all the subarrays end up with approximately equal size.)

8.1 Which order statistics do we use?

The problem requires us to split the array A into k subarrays such that the sizes of any two differ by at most 1. How can we do this? One obvious answer is to define a step size $S = \lfloor n/k \rfloor$ and then split the array at locations $S, 2S, 3S, \ldots, (n-1)S$. This does not quite work; for example, when $n = 17$ and $k = 3$, the splits occur at 5 and 10, and our intervals are 1..5, 6..10, 11..17 (this last one is too large). A simple tweak is to split at indices $\lfloor n/k \rfloor, \lfloor 2n/k \rfloor, \lfloor 3n/k \rfloor, \ldots, \lfloor (k-1)n/k \rfloor$ instead. This will work, but there is another simple way that is more obviously correct. Let $q = \lfloor n/k \rfloor$ and $r = n \bmod k$. In our example, $n = 17, k = 3, q = 5, r = 2$. Write the number q out k times:

$$5, 5, 5.$$

Then add 1 to the first r numbers:

$$6, 6, 5.$$

These are the sizes of the subarrays we want. Now take the partial sums:[4]

$$6, 12, 17.$$

[3] unless $s < M[1]$, but this case is easily handled. For instance, we can define $M[0] = 0$.

[4] The i-th partial sum is the sum of the first i terms.

These are the order statistics we want to find. By construction, the sizes differ by no more than one, since the difference between any two consecutive indices is either q or $q+1$. In this example, our intervals will be 1..6, 7..12, 13..17. As a check, we see that their sizes are 6, 6, 5.

There are, of course, other ways to do this, but the way we have just described is simple to implement, easily justified, and runs quickly (it only takes $O(k)$ time).

Proof of Lemma 2

Let H be a heap on n nodes with n even. When $n = 0$ the statement is clearly true. For $n > 0$ we will use induction. We assume the following induction hypothesis: *The statement holds true for a heap on m nodes, for all even values of m less than n.* Let H_L and H_R be the left and right subtrees of the root of H. One of them has an even number of nodes, the other odd. Say H_L has an even number. By the induction hypothesis, there is a suitable matching for H_L. Now H_R has an odd number. Remove its last leaf to create H'_R. By the induction hypothesis, there is a suitable matching on H'_R. We now replace the leaf, and we match it to the root of H. Now every node in H is matched.

If the subtree with an even number of nodes was in fact H_R, the proof will still work after switching the roles of H_L and H_R. □

Proof of Lemma 3

Let $\ell_1, \ldots, \ell_{n/2}$ be the leaf values of H in increasing order, and let $b_1, \ldots, b_{n/2}$ be the branch values corresponding to each matched leaf. Therefore, the leaf node containing ℓ_i is matched to the branch node containing b_i, for all $i = 1, \ldots, n/2$.

Write the elements in the order

$$\ell_1, b_1, \ell_2, b_2, \ldots, \ell_{n/2}, b_{n/2}.$$

We want to put these into increasing order. We can do this by moving each b_i to the right, into its proper place. (We never have to move any b_i left, because $\ell_i < b_i$. The b_i's may be out of order with one another, but we can fix this using all right moves; left moves are never necessary.) As we do this, we focus our attention on the right half of the list, which contains $n/2$ elements, and keep track of how many b_k's it contains. Each time we move a b_i into the right half of the list, we evict one of the members (the lowest one). The member we evict may be a b_k or an ℓ_k: in the former case the number of b_k's in the right half stays the same, and in the latter case the number increases by 1. It is evident that this number never decreases. Initially the number was $\left\lceil \dfrac{n/2}{2} \right\rceil = \lceil n/4 \rceil$.

When we are done, the median of H fits into the ordering exactly in the middle. Therefore, at least $\lceil n/4 \rceil$ branch nodes are greater than the median. There are $\lfloor n/2 \rfloor$ branch nodes altogether, so that those at least $\lceil n/4 \rceil$ branch nodes are at least half of all of the branch nodes of H. □