CS521 Fall 2011 \ Assignment #2

Ariel Stolerman

1)

Proof by induction (upside down: base case is top most h, inductive step is from height h to height $h-1$)

Let n be the size of the heap, so the height of the heap will be $\lceil \lg n \rceil$ - the root is at the top most level, and since a heap is a left filled binary tree, all levels but the last are completely full and the last level will have between 1 and until completely full nodes. The *ceiling* notation denotes we have a discrete number of levels, and rounding up accounts for a whole level even if it's not completely full.

The root of the heap is a single node at the top most level, that is $h = \lceil \lg n \rceil$. Let $m = \begin{cases} n, & \lg n = \lceil \lg n \rceil \\ 2^{\lceil \lg n \rceil}, & \lg n < \lg \lceil n \rceil \end{cases}$ – the number of nodes in the root is indeed $\left\lceil \frac{n}{2^{\lceil \lg n \rceil + 1}} \right\rceil = \left\lceil \frac{n}{2m} \right\rceil \underset{m \geq n}{=} 1$.

Now we will assume that the above is true down to and including height $h+1$, and prove for height h:

If the level at height h has any elements, that means the level at height $h+1$ is not the last one, so by the heap conditions it must be full, and by the inductive assumption it has exactly $\left\lceil \frac{n}{2^{(h+1)+1}} \right\rceil$ nodes (*at most* if it was the last level) which are $\left\lceil \frac{n}{2^{h+2}} \right\rceil$ nodes. The level beneath it at height h will have:

- Exactly twice as much nodes if it is also not the last level (by the heap conditions).
- At most twice as much nodes.

Thus the number of nodes at height h will be at most $2\left\lceil \frac{n}{2^{h+2}} \right\rceil = \left\lceil \frac{2n}{2^{h+2}} \right\rceil = \left\lceil \frac{n}{2^{h+1}} \right\rceil$, as required.

2)

If A is an array of size n, the running time of *HeapSort* for an initial A both sorted in increasing order and decreasing order is $\Theta(n \lg n)$. The reason for that is:

- The *build-max-heap* procedure runs in $O(n)$ time in both cases, as we proved with the amortized analysis in class. When the array is in a decreasing order no actual swap has to be taken since the array already satisfies the max-heap condition, but it is still $O(n)$ (going through the elements and comparing them with their children).
- The second phase starts in both cases when all nodes at each level are bigger than all nodes at the level beneath it. That means each swap of the root (max) with the last node in the heap will result with heapifying the new root down to the maximum. Each step we remove the maximum, decreasing the size of the heap by 1. That results with each node that was put instead of the maximum being heapified down $\lceil \lg(size - of - heap) \rceil$ levels, which means a total of $\sim \sum(\lg i) = \Theta(n \lg n)$.

To conclude, in both cases due to the heapify operations the running time is $\Theta(n \lg n)$.

3)

Let $l_1, \dots, l_k$ be k sorted lists of sizes $n_1, \dots, n_k$ such that $\sum_{i=1}^{k} n_i = n$. Here is an algorithm to merge all lists (keeping the new sequence sorted) in $O(n \lg k)$ time:

First, note that: $1 \leq k \leq n$ (there is at least 1 list and at most n lists).

We will assume the lists are sorted in a decreasing order, and that there are three $O(1)$ procedures that can be applied on any list:

- *top*: returns the value of the element at the top of the list (the maximum).
- *pop*: removes the element at the top of the list and returns its value (the maximum).
- *size*: returns the size of the list.

The algorithm:

1. Let *result* be a new empty list.
2. Build a Heap of size k called *list-heap* from all lists $l_1, \dots, l_k$, looking at their *top* as the list's representing value.
3. For $i = 1$ to n:
    3.1. If the size of the top-most list is 1:
        3.1.1. Remove the list, pop it and append the value to *result*.
        3.1.2. Put the last list in the heap at the root instead.
        3.1.3. Update the heap size to be -1.
    3.2. Otherwise:
        3.2.1. Pop the top-most list and append the value to *result*.
    3.3. run *max-heapify(list-heap, 1)*
4. Return *result*.

Correctness:

Since all lists are sorted in a decreasing order, the maximum of all the top (maximum) values of all lists is the absolute maximum. Therefore after the initial list-heap build and after every iteration in phase 3, the maximum of the top-most list is the absolute maximum – and smaller than all extracted maximum so far – so the result list is built in a decreasing order (this could be more rigorously shown by induction).

The correctness of the heap-list condition (the top of the top-most list is the absolute maximum of the heap) is kept along the iterations because:

- If 3.1 applies, we completely remove a list from the heap, and the correctness is kept just like in a normal heap – the last element put at the root makes sure the heap is kept left-filled, and heapifying it down keeps the heap condition, so the top of the top-most list is the absolute maximum.
- If 3.2 applies, after popping the top-most list we heapify it down, thus keeping the heap condition.

Running n iterations makes sure we get all elements in all lists.

Therefore we eventually get all n elements sorted in a decreasing order in the result list *result*.

Running time:

1. Creating a new empty list is $O(1)$.
2. Building the *list-heap* given *top* costs $O(1)$ and k lists is $O(k) \underset{k \leq n}{=} O(n)$ (as shown in class).
3. Each of the n iterations costs $O(\lg k)$, as this is the cost of *max-heapify* in a k-size heap ⇒ the total is $O(n \lg k)$.

⇒ the total running time is $O(n \lg k)$, as required.

4)

I will interpret an almost-sorted input for the problem presented in the question as follows: each element in the input array is in an environment of elements such that it might not be sorted in that environment, but it is greater than all elements to the left of that environment, and less than all elements to the right of that environment. And most importantly, the size of that environment is of a constant size. In other words:

For $A = < a_1, \dots, a_n >: \forall a_i \exists\ constant\ k_i\ s.t.: \begin{cases} \forall j < i - k_i : a_j \leq a_i \\ \forall l > i + k_i : a_l \geq a_i \end{cases}$

In that case, the running time of **insertion sort** will be n times × some constant k (could be $\max\{k_i\}$) since no more than that number of comparisons (and swaps) is required, resulting in $O(n)$.

However, using **quicksort** even with the best constant possible (derived from the best division – when the pivot is selected as the median for each split) is still $\Omega(n \lg n)$. To be more precise, in this case since we have n/k sorted bins, each containing k unsorted elements, each iteration in quicksort will divide the next phase into one recursive call over a constant number of elements (at most $k - 1$ to be exact, to the left of the chosen pivot) and the other recursive call would be over $n - ik$ elements, i being the level of recursion. Since it is not a portion on n on each side of the pivot, as seen in class, this would result in a $O(n^2)$ running time.

Therefore, under some assumptions regarding the exact constants that play here, using insertion sort for an almost-sorted input would be more efficient than using quicksort.

5)

The proof that the running time of the suggested algorithm is $O\left(nk + n \lg\left(\frac{n}{k}\right)\right)$ has similar arguments as presented in the answer to question #4.

The "quicksort" part of the algorithm costs as follows:

- The total cost for each level of the recursion is n, since in total for each level i we have 2^i calls of quicksort on inputs of sizes $n/2^i$, thus resulting in n for each level.
- We continue the recursion until we get bins of size k, thus the height of the recursion tree would be as if we were running "regular" quicksort on an input of size $\frac{n}{k}$. Therefore the height of the tree is $\lg\left(\frac{n}{k}\right)$.

That concludes to a total cost of this part of $O\left(n \lg\left(\frac{n}{k}\right)\right)$.

Now the "insertion sort" part runs on an array divided into $\frac{n}{k}$ bins of size k such that there's a total order between the bins (meaning all elements in bin i are greater than all elements in all bins $1, \dots, i-1$), but no order within each bin is promised. That means that the insertion sort will move each of the n elements at most k places (and perform k comparisons), resulting in a total cost of $O(nk)$.

The total cost is therefore $O\left(nk + n\lg\left(\frac{n}{k}\right)\right)$.

In theory k should be picked in a way that it will satisfy $c_1\left(nk + n\lg\left(\frac{n}{k}\right)\right) \leq c_2(n\lg n)$, c_1, c_2 being the constants derived from the exact implementation of the quicksort and insertion sort procedures. In practice a series of experiments should be performed to set the threshold, taking into consideration other factors such as machine characteristics and expected input size.

6)

If the *select* algorithm used a division into groups of 7 elements, it would still be linear. Compared to the analysis in the original algorithm, where (in the book) $T(n) = T(\lceil n/5\rceil) + T(7n/10 + 6) + O(n)$, here:

If we divide n into groups of size 7 we will get $\left\lceil\frac{7}{n}\right\rceil$ groups. After finding the median of medians, we know in a similar manner to the original argument that at least half of the $\left\lceil\frac{7}{n}\right\rceil$ groups contribute at least **4** elements that are less than x, except for the group with fewer than 7 elements (if n is not divisible by 7) and the group that contains x. So the number of elements that are less than x is $4\left(\left\lceil\frac{1}{2}\left\lceil\frac{n}{7}\right\rceil\right\rceil - 2\right) \geq \frac{4n}{14} - 8 = \frac{2n}{7} - 8 \Rightarrow$ in worst case the recursive call to *select* will be on at most $\frac{5n}{7} + 8$.

Therefore the recurrence is $T(n) = T(\lceil n/7\rceil) + t(5n/7 + 8) + O(n)$. Now by substitution we will show an upper bound of cn, proving at most linear running time:

$T(n) \leq c\lceil n/7\rceil + c(5n/7 + 8) + an$ / a is the constant for $O(n)$

$\leq cn/5 + c + 5cn/7 + 8c + an$

$= 6cn/7 + 9c + an$

$= cn + (-cn/7 + 9c + an)$ $/ -cn/7 + 9c + an \leq 0 \Rightarrow c \geq 7a \cdot \frac{n}{n-63} \Rightarrow choose: \begin{cases} n > 126 \\ c \geq 14a \end{cases}$

$\leq cn$

So for $n \geq 126$ and choosing $c \geq 14a$ we get $T(n) = O(n)$, as required.

If we use division to groups of 3, the algorithm will not be linear. The intuition for that is that for 3 we get that the number of elements less than x is: $2\left(\left\lceil\frac{1}{2}\left\lceil\frac{n}{3}\right\rceil\right\rceil - 2\right) \geq \frac{n}{3} - 4 \Rightarrow$ there are $\frac{2n}{3} + 4$ in worst case for the recursive calls. That concludes to the recurrence: $T(n) = T(\lceil n/3\rceil) + T(2n/3 + 4) + O(n)$. But the total sum of the recursive input of the recursive calls is more than n, giving a hint that the $O(n)$ won't "take on" the recursive calls, giving a larger running time.

Using substitution we show that $T(n) = \Omega(n\lg n)$:

$T(n) = c\left\lceil\frac{n}{3}\right\rceil \cdot \lg\left\lceil\frac{n}{3}\right\rceil + c\left(\frac{2n}{3}+4\right)\lg\left(\frac{2n}{3}+4\right) + an$ / a is the constant for $O(n)$

$\geq \frac{cn}{3}(\lg n - \lg 3) + \left(\frac{2cn}{3}+4c\right)(\lg 2 + \lg(n+6) - \lg 3) + an$ / removing $4c$ and $\lg 2$ lowers the sum

/ $\lg(n+6) > \lg n$

$\geq \frac{1}{3}cn\lg n - \frac{1}{3}cn\lg 3 + \frac{2}{3}cn\lg n - \frac{2}{3}cn\lg 3 + an$

$= cn\lg n - cn\lg 3 + an$ / choose $c \leq \frac{a}{\lg 3} \Rightarrow (-cn\lg 3 + an) \geq 0$

$\geq cn\lg n$

$\Rightarrow T(n) = \Omega(n\lg n)$, so definitely $T(n) \neq O(n)$ when dividing to groups of 3.

7)

Here is an algorithm in $O(n)$ running time to find the k closest (**by position, not value**) number to the median of a set S of distinct numbers (we address the lower median as the median):

The algorithm:

1. Let M be an empty set to contain the required output.
2. Let A be an array containing the elements in S, and $n = A.length$.
3. Find $m = select(A, 1, n, \lfloor n/2 \rfloor)$ – the (lower) median of A.
4. Run $partition(A, m)$, set the left to be A_1 of length n_1 and the right to be A_2 of length n_2.
5. If k is even: find $l = select(A_1, 1, n_1, n_1 - k/2)$ and $r = select(A_2, 1, n_2, k/2)$
6. Otherwise:
    a. Find $l = select(A_1, 1, n_1, n_1 - (k-1)/2)$ and $r = select(A_2, 1, n_2, (k-1)/2)$
    b. Randomly select either $select(A, 1, n_1, n_1 - (k-1)/2 - 1)$ or $select(A, 1, n_2, (k-1)/2 + 1)$ and add to M.
7. Find all elements between l and r and add them to M: run partition on A_1, l and A_2, r and add to M all elements in A_1 that are bigger than or equal to l and all elements in A_2 that are less than or equal to r.
8. Return M.

Since we are using a set of distinct numbers, we know the partitions we run would give us exactly $k/2$ (or $(k-1)/2$) elements to be added to M for each side of the median. We take care of the extra element in the case of an odd k by taking one randomly, and since they both have the same distance from the median it doesn't matter which. Each procedure ran in the algorithm is either constant or linear, concluding in a total $O(n)$ running time.

Note: the algorithm below is for the case we are required to return the k closest **by value**. Skip it if it's unnecessary.

First we define the following procedures:

- $a.normalize(m)$: sets $a = \begin{cases} a, & a \leq m \\ a - 2(a-m), & a > m \end{cases}$ – if a is larger than m, it moves it to be at the same distance of m but from below.
- $a.true - value$: if a was normalized, it sets it back to what it was and returns it. Otherwise it just returns it.

The algorithm:

1. Let M be an empty set to contain the required output.
2. Let A be an array containing the elements in S, and $n = A.length$.
3. Find $m = select(A, 1, n, \lfloor n/2 \rfloor)$ – the (lower) median of A.
4. Run $partition(A, m)$
5. For each $a \in A\ s.t.\ a > m$: $a.normalize(m)$ – finding all such a's is simply taking the right side of m in A after the partition.
6. Remove m from A.
7. Let $x = select(A, 1, n-1, n-k-1)$
8. Run $partition(A, x)$
9. For each $a \in A\ s.t.\ a > x$: put $a.true - value$ in M
10. Return M.

What the algorithm does is finding the median, normalizing all elements to be lower than it but keeping the original distance from the median, and then finding the $n-k$ order statistic of this new array. Surely all the elements that are larger than that order statistic, taking their original non-normalized value, are the ones closest to the median (since all numbers are distinct, we won't get "holes", but I won't elaborate on that here).

Running time: every step of the 10 steps above is $O(n)$, thus concluding to a $O(n)$ running time, as required.

8)

The intuition to find an algorithm in the required running time is that it would be a recursive algorithm such that the height of the recursion tree is $\lg k$, thus dividing the task into two completing tasks each level of the recursion.

Here is an algorithm to find the $k-1$ kth quantiles for an input array A:

1. Let $k - quantiles$ be an empty list to accumulate the quantiles.
2. Define $Find - k - quantiles(A, k)$ as follows:
   2.1. If $k > 1$:
      2.1.1. Let $n = A.length$
      2.1.2. Let $i = \lfloor k/2 \rfloor \times \frac{n}{k}$
      2.1.3. Find $pivot = select(A, 1, n, \lfloor i \rfloor)$ and add it to $k - quantiles$.
      2.1.4. Run $partition(A, pivot)$, set A_1 to be the left side including the $pivot$ and A_2 to be the right side.
      2.1.5. Call $Find - k - quantiles(A_1, \lfloor k/2 \rfloor)$ // find all kth quantiles to the left
      2.1.6. Call $Find - k - quantiles(A_2, \lfloor k/2 \rfloor)$ // find all kth quantiles to the right
3. If required sort $k - quantiles$.

Each iteration finds the median kth quantile of the set of k quantiles that is contained in this iteration's input array (as in the book, we take the lower median in case it is out of an even-sized set). For instance, the first iteration finds the median

of the total set of kth quantiles; the first recursive call under that iteration finds the median of the left half of the kth quantiles set, and the second recursive call finds the median of the right half of the kth quantiles set, and so on.
That means that each iteration partitions the input into 2 inputs for the next calls that each contains at most half of the kth quantiles that are contained in the current iteration. This process will then repeat $\lg k$ times – as each recursive call contains twice as less kth quantiles as the call before it. In addition, every call costs $O(size\ of\ input\ array)$, as the *select* and *partition* procedures are linear in their input, plus some constant time operations. Each level has input array twice as less as the level before it and each level has twice as much calls as the level before it – thus concluding to $O(n)$ total cost per level. That concludes to a total cost of $O(n \lg k)$, as required.