# CS 521: Data Structures and Algorithms I
# Homework 2

Dustin Ingram

October 18, 2011

1. **Solution:** In a full binary heap, every node (excluding the lowest level where $h = 0$) has two children, thus, the number of nodes at any given height $h$ is twice that of it's preceding height $h+1$. Inductively, this means that each new level doubles the number of nodes in the tree, i.e., for a full tree of $n$ nodes, the last level will contain $\lceil n/2 \rceil$ of the nodes. Since $h$ (in this instance) is defined as the distance from the lowest level of the tree, this means that at any height $h$ there are $\lceil n/2^{h+1} \rceil$ nodes.

2. **Solution:** The original order of the array $A$ of length $n$ has no effect on the running time of Heapsort, since for every $n$ elements in the array, Max-Heapify, which makes at most $\lg n$ comparisons, must be called once, for a total running-time of $n \lg n$ in both cases.

3. **Solution:** Since each of the $k$ lists are already sorted, two lists can be merged into one sorted list in linear time, regardless of size. If, at each iteration of the algorithm, we merge $k$ lists of size $n/k$ into $k/2$ lists of size $2n/k$ via pairwise merging, for a total of $k * (n/k) = n$ comparisons, we can produce a single sorted list in $lg(k)$ iterations, giving the algorithm a overall runtime of $O(n \lg k)$.

4. **Solution:** The complexity of correctly placing a single element in a list using Insertion-Sort depends directly on the number of adjacent elements the chosen element must be compared with. In a usual case of a initially random array of size $n$, this may be as many as $n$ elements, producing a worst-case performance of $O(n^2)$. However, if we can assure that the maximum number of comparisons to adjacent elements will not exceed some relatively small constant $k$, as in the case of check-sorting, the performance can be improved to, at most, $O(nk)$.

   In the case of Quicksort, however, the single element's proximity to it's correct positioning does not improve the complexity of the algorithm, which still needs to perform $\lg n$ comparisons for every $n$ elements for a total complexity of $n \lg n$. As long as the constant $c < \lg n$, the Insertion-Sort algorithm will out-perform Quicksort for cases of almost-sorted input.

5. **Solution:** Continuing from the previous solution, we see that if we allow Quicksort to return a list full of nearly-sorted sub-arrays of at most size $k$, that running Insertion-Sort then guarantees a additional complexity of $O(nk)$ due to the aforementioned relative adjacency. Stopping Quicksort prematurely reduces the number of required partitioned iterations from $\lg n$ to $\lg n/k$ (because this is, essentially, sorting $n/k$ unsorted sub-arrays of size $k$ as singular elements), thereby reducing the complexity to $O(n \lg n/k)$. Combining these two operations as one simply results in a combined running time of $O(nk + n \lg n/k)$.

   For this hybrid algorithm to be successful, Insertion-Sort must be able to out-perform Quicksort; As mentioned in the previous solution, this only occurs when $k < \lg n$, so realistically, $k = \lfloor \lg n \rfloor$.

6. **Solution:** The Select algorithm will still work in linear time for groups of 7, or rather, for any odd number of groups $\geq 5$. If groups of 5 are selected, the number of elements greater than the median is as follows:

$$3\left(\left\lceil \frac{1}{2}\left\lceil \frac{n}{5} \right\rceil \right\rceil - 2\right) \geq \frac{3n}{10} - 6$$

   Therefore, Select would be called recursively on $7n/10 + 6$ elements, resulting in the following recurrence:

$$\begin{aligned} T(n) &\leq c\lceil n/5 \rceil + c(7n/10 + 6) + an \\ &= cn/5 + c + 7cn/10 + 6c + an \\ &= 9cn/10 + 7c + an \\ &= cn + (-cn/10 + 7c + an) \end{aligned}$$

   Which will not exceed $cn$ if

$$\begin{aligned} 0 &\geq -cn/10 + 7c + an \\ c &\geq 10a(n/(n - 70)) \end{aligned}$$

   Similarly, if groups of 7 are selected, at least half of the groups contribute at least 4 elements greater than the median, so the number of elements greater than the median is as follows:

$$4\left(\left\lceil \frac{1}{2}\left\lceil \frac{n}{7} \right\rceil \right\rceil - 2\right) \geq \frac{2n}{7} - 8$$

   Therefore, Select would be called recursively on $5n/7 + 8$ elements, resulting in the following recurrence:

$$\begin{aligned} T(n) &\leq c\lceil n/7 \rceil + c(5n/7 + 8) + an \\ &= cn/7 + c + 5cn/7 + 8c + an \\ &= 6cn/7 + 9c + an \\ &= cn + (-cn/7 + 9c + an) \end{aligned}$$

Which will not exceed $cn$ if

$$\begin{aligned} 0 &\geq -cn/7 + 9c + an \\ c &\geq 7a(n/(n-63)) \end{aligned}$$

For which a suitable $c$ can be chosen, and the algorithm will run in linear time. However, if groups of 3 are selected instead, this does not hold true. In this case, the number of elements greater than the median is as follows:

$$2\left(\left\lceil \frac{1}{2}\left\lceil \frac{n}{3} \right\rceil \right\rceil - 2\right) \geq \frac{n}{3} - 4$$

Therefore, Select would be called recursively on $2n/3 + 4$ elements, resulting in the following recurrence:

$$\begin{aligned} T(n) &\leq c\lceil n/3 \rceil + c(2n/3 + 4) + an \\ &= cn/3 + c + 2cn/3 + 4c + an \\ &= cn + 5c + an \\ &= cn + (5c + an) \end{aligned}$$

For which $c \geq -an/5$, and there is no linear solution to the recurrence.

7. **Solution:** The median of a set of $n$ elements is the $\left(\frac{n}{2}\right)$-th order statistic; the $k$ elements closest to this median, therefore, are bounded by the $\left(\frac{n}{2} - \frac{k}{2}\right)$-th order statistic and the $\left(\frac{n}{2} + \frac{k}{2}\right)$-th order statistic, each of which can be found using Select in linear time. Next, the unsorted array $A$ can then be linearly scanned, and the algorithm can return any elements within this bound which are not the median, which is also found in linear time as the $\left(\frac{n}{2}\right)$-th order statistic.

8. **Solution:** It is trivial to use Select to individually determine each of the $i \cdot n/k$-th quantiles for $i = \{1, \ldots, (k-1)\}$, however, such a solution would run with $O(nk)$ complexity (that is, using Select results in $O(n)$ complexity for every $k$ quantiles). However, by partitioning the $n$ elements each time a quantile is found, we can instead reduce the number of elements which Select must be called on by half every iteration:

   - Iteration 1: We find the $\lfloor k/2 * n/k \rfloor$-th order-statistic of the complete array of $n$ elements for a cost of $O(n)$ and partition, repeating the process on each half;
   - Iteration 2: We find the $\lfloor k/4 * n/k \rfloor$-th order-statistic of each array of $n/2$ elements for a total cost of $O(n/2) + O(n/2) = O(n)$ and partition, repeating the process on each half;
   - Iteration $i$: We find the $\lfloor k/i * n/k \rfloor$-th order-statistic of each array of $n/i$ elements for a total cost of $i \cdot O(n/i) = O(n)$, until $k/i = 1$.

   Since $i$ is growing at a rate of $2i$ every iteration, the recursion tree is binary and has a height of $\lg k$; since the cost of each iteration sums to $O(n)$, this results in the desired complexity of $O(n \lg k)$.