1. (a) Let us assume that the root is stored at array position $A[1]$. Any node can be uniquely described by its level number (root has level number 0) and its number from the left within that level. Thus a node may be at level $l$ and may be the $j^{th}$ node from the left, in that level. The index of that element in the array can then be calculated as $j +$ total number of nodes up to level $l - 1$. This is equal to $j + \sum_{t=0}^{l-1} k^t = j + \frac{k^l - 1}{k - 1}$. The index of its $i^{th}$ child is $= \text{total number of nodes up to and including level } l + (j - 1)k + i = \sum_{t=0}^{l} k^t + (j - 1)k + i = \frac{k^l - 1}{k - 1} + jk - k + i = k \left( j + \frac{k^{l-1} - 1}{k - 1} \right) + (1 - k) + i = k \times \text{index of node} + (1 - k + i)$.

It follows immediately, that the parent of a node of index $x$ is $\lceil \frac{x - 1}{k} \rceil$.

- Here we vary the index from the last internal node down to the node with index 1, which is the root. If we traverse the nodes in this order, and convert them into heaps, then every time we arrive at a node the subtrees rooted at its $k$ children are all heaps. The fact that we start at the last internal node, means that all its children are leaves and hence, by default are one element heaps. We perform a heapify operation by swapping the element at a node with the maximum among its children whenever it has a value less than that maximum. We move it down the tree until it is greater than all its children. At each node, the heap property is satisfied at that node. Thus at the end the heap property is satisfied at the root.

The total cost can be bounded by the total number of atomic
heapify operations. The cost of each atomic heapify (ignoring further recursive calls) is $O(k)$, since we need to find the maximum of $k+1$ elements. All nodes from the level above the leaf potentially call atomic heapifies at the nodes of height 1 (parents of leaves). All nodes of height at least 2 call atomic heapifies at nodes of height 2 and so on. Thus the number of calls to atomic heapify is at most $\left\lceil \frac{n-1}{k} \right\rceil + \left\lceil \frac{n-1}{k} \right\rceil /k + \cdots$. The cost is thus $O \left( k \left( \left\lceil \frac{n-1}{k} \right\rceil + \left\lceil \frac{n-1}{k} \right\rceil /k + \cdots \right) \right) = O(n)$.

- We can also restrict the heapseize to be one and incrementally grow the size of the heap by one at a time. Everytime we increase the size of the heap, we ensure the new node also satisfies the heap property. We begin with only the root, which is trivially a heap and increase the heapseize until the whole array is a heap. When we come to node $i$, the entire array up to node $i-1$ is a heap. Thus the only potential violation is between node $i$ and its parent, which we have already seen has index $\left\lceil \frac{i-1}{k} \right\rceil$. We need only one comparison between the node and its parent and if the node has value greater than the parent’s we simply exchange the keys and recurse upwards. Since the node was greater than all its children, prior to the expansion of the heap, this exchange ensures that the heap property is satisfied now at the parent. Again, we make a similar comparison upwards till there is no violation or we reach the root. Each atomic heapify takes $O(1)$ time independent of $k$. This time the number of atomic calls to heapify at nodes of height 1 is $n - \left\lceil \frac{n-1}{k} \right\rceil$. The number of atomic calls to heapify at nodes of height 2 is $n - \left\lceil \frac{n-1}{k} \right\rceil - \left\lceil \frac{n-1}{k} \right\rceil /k$. These values are approximately $\frac{k-1}{k}n$, $\frac{k^2-1}{k}n$ etc. Thus, the total cost is this cumulative cost of these terms, of which there are approximately $\log_k n$. The terms approach closer and closer to $n$. Thus the total cost is roughly $n \log_k n$ which is asymptotically worse than the previous approach. Thus, although there is a saving in the cost of an atomic heapify, this is more than offset by the total number of calls to atomic heapify.

(b) We assume it is a maximum heap. In this case, if the key value is increased, any violation can only be with the parent. In this case, we exchange the key of the node with that of the parent
and move upwards until either the violation ends or the root is reached. Thus the total cost of this operation is $O(\log k n)$. In case of the key value being decreased, the violation could only be with one of the children. Determining which child to exchange with, if any takes $O(k)$ time and this will have to be repeated till the violation ends or a leaf is reached. Thus the total cost is $O(k \log k n)$.

The $\log k n$ terms in the above expressions can be replaced more precisely by the depth and the height of the nodes respectively.

(c) If the node is the last node of the heap, then we merely need to reduce the heapsize by one. Otherwise, we copy the last element of the heap into the index of the deleted node and reduce the heapsize by one. If the key of the copied node is greater than the original key, then we heapify in the upward direction and otherwise we restore the heapify in the downward direction as described in part (b) above. The complexity is again the same as in those procedures.

(d) We place the element into the last position of the heap after increasing the heapsize by one. We then move the new element up the heap until either there is no heap property violation with the parent or it reaches the root. The cost is again $O(\log k n)$.

2. Modify randomised quicksort to pick a set of three elements uniformly at random and then partition around the median element. We assume that the number $n$ of elements is at least 3, and that the array contains all distinct elements.

(a) The probability of picking an element of rank $i$ as pivot by this method is $= \frac{(i-1)(n-i)}{n(n-1)(n-2)/6} = \frac{6(i-1)(n-i)}{n(n-1)(n-2)}$. This is basically obtained by taking the ratio of three element subsets in which the element of rank $i$ is the median to the total number of three element subsets.

(b) A split at least as balanced as $\alpha, 1 - \alpha$ is obtained whenever the pivot element is in this fractional range. Thus, the probability of such a split is

$$= \sum_{i=\lceil n\alpha \rceil}^{\lfloor n(1-\alpha) \rfloor} \frac{6(i-1)(n-i)}{n(n-1)(n-2)}$$
\[
\frac{(n-n\alpha)}{\sum_{i=1}^{n(n-1)(n-2)}} \frac{6(i-1)(n-i)}{n(n-1)(n-2)} - \sum_{i=1}^{(n\alpha)} \frac{6(i-1)(n-i)}{n(n-1)(n-2)}
\]

\[
= \frac{6}{n(n-1)(n-2)} \sum_{i=1}^{(n-n\alpha)} ((n+1)i-n-i^2) - \sum_{i=1}^{(n\alpha)} ((n+1)i-n-i^2)
\]

\[
= \left[ \frac{(n-2\alpha)(n^2-9n-4-2a^2+4n+3\alpha)+a((\alpha+1)(2\alpha+1)-(n-\alpha+1)(2n-2\alpha+1))}{n(n-1)(n-2)} \right]
\]

Intuitively this method should yield a better balance than just picking one element. This can be verified because for the same value of \(\alpha\), the probability of getting a correspondingly balanced split can be seen to be higher by substituting in the formula derived above.

(c) The running time of Randomised Quick Sort is \(\Omega(n + X)\), where \(X\) is the number of comparisons between elements of the input sequence. We use a set of indicator random variables \(X_{ij}\) which is 1 precisely when the \(i^{th}\) and the \(j^{th}\) elements are compared in the particular execution of randomised quick sort and 0 otherwise. Thus, the expected running time, \(E[T(n)] = E[\Omega(n + X)] = \Omega(n) + E[\Omega(X)]\). The random variable \(X\) is actually

\[
\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}
\]

Hence, the expected running time is \(\Omega(n) + \Omega(E[X])\).

\[
\Omega(E[X]) = \Omega\left( \mathbb{E} \left[ \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij} \right] \right)
\]

The expected value \(E[X_{ij}] = 1 \times Pr(X_{ij} = 1) + 0 \times Pr(X_{ij} = 0) = Pr(X_{ij} = 1) = \frac{2}{j-i+1}\). Thus,

\[
E[T(n)] = \Omega(n) + \Omega \left( \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \right)
\]

\[
\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \geq \sum_{i=1}^{(n/2)} \frac{2}{k} = \Omega(n \log n)
\]

Thus \(E[T(n)] = \Omega(n \log n)\).