Question 1: Randomized Quicksort

1.1 A

In the worst case Randomized Quicksort would be $O(n^2)$, since, just as in regular Quicksort, if the randomly chosen pivot happens to be either the min or the max of the data set, then the recursion would produce a skewed tree which is $O(n^2)$

1.2 B

Note: Proof idea taken from Tufts Comp160 video lecture on Quicksort posted online.

To find the expected runtime of a randomized Quicksort we take a look at the number of expected comparisons we will make. Let X represent our data set, with X_i and X_j representing the ith and jth elements of the set respectively. Let X_{ij} represent the set of elements between i and j inclusively. If a randomly chosen pivot occurs in the set X_{ij} then X_i and X_j will be sorted into different partitions and thus will never be compared. So two random element X_i and X_j will only be compared if a pivot is chosen that is either X_i or X_j.

So we want the probability that X_i or X_j is the first pivot chosen. Let $P(Y)$ represent that probability. Let $P(X_i)$ be the probability that X_i is the first pivot chosen and $P(X_j)$ be the probability that X_j is the first pivot chosen. So $P(Y) = P(X_i)+P(X_j)$. The probability of both $P(X_i)$ and $P(X_j)$ is 1 over the total number of elements in the set X_{ij}. So $P(X_i)\&P(X_j) = \frac{1}{j-i+1}$.

$$P(Y) = \frac{2}{j-i+1}$$

Let $E[X]$ represent the expected number of comparisons made.

$$\begin{aligned} E[X] &= \Sigma_{i=1}^{n-1}\Sigma_{j=i+1}^{n} P(Y) \\ &= \Sigma_{i=1}^{n-1}\Sigma_{j=i+1}^{n} \tfrac{2}{j-i+1} \\ &= \Sigma_{i=1}^{n-1}\Sigma_{k=1}^{n} \tfrac{2}{k} \\ &= \Sigma_{i=1}^{n-1} O(\log n) \\ &= O(n \log n) \leftarrow \text{expected runtime of Randomized Quicksort} \end{aligned}$$

1.3 C

In order to guarantee that Randomized Quicksort runs in $O(n \log n)$ in the worst case, you need to eliminate the possibility that the pivot chosen is in the min or the max of the set. You can ensure that this doesn't happen by selecting the median element as the pivot. Median finding takes $O(n)$ time and so the worst case would be $O(n \log n)$.

1.4 D

Let our set of elements be: $\{1_1, 3_2, 1_3, 4_4\}$ with subscripts denoting start positions in the array. A stable sorting algorithm ensures that after elements are sorted equal elements appear in the same order as pre-sorting. So a stable algorithm would return $\{1_1, 1_3, 3_2, 4_4\}$. The 1_1 occurs before 1_3 in both the original and sorted set.

However, choosing 1_1 as the pivot will produce the following sequence $\{1_3, 1_1, 3_2, 4_4\}$. The 1_1 occurs after 1_3 in the sorted set and therefore Randomized Quicksort is not stable.

Question 2: Hashing

Modifying hash chaining to keep elements in increasing sorted order as opposed to arrival order would have the following effects on these functions:

2.1 Insertion

Insertion would theoretically take a little less time for very long chains. If we are inserting in arrival order, we insert at the end of the chain each time, which means perusing through the linked list until the end each time. If we insert based on increasing order, we insert at the end of the chain in the worst case, on average we will not be traversing the entire list.

2.2 Deletion

Deletion, assuming the chain is doubly linked, I wouldn't think would be any different. Since you have already found the key with a successful search, you just need to delete that element, and it is unknown where that element lies in the chain in either case, the result would be the same.

2.3 Successful Search

Successful Search would be unchanged as well. You don't know how the element you're looking for relates to the rest of the chain. It could be anywhere in both types of chaining.

2.4 Unsuccessful Search

Unsuccessful Search however is faster. Since you know the elements are in increasing order, if the element you're looking for is smaller than the element you're at in the chain, regardless of how long the chain still is you know it is not present in the table.