1. (a) **FALSE.** Here is a counterexample: Let \( f(n) = n \) and \( g(n) = 2n \). Clearly, \( f(n) = \Theta(g(n)) \). However, \( n! = o((2n)!)) \).

(b) **FALSE.** \((\log n)^{\sqrt{n}} = 2^{\sqrt{n} \log \log n} \). On the other hand, \((\sqrt{n})^{\log n} = 2^{(1/2)\log^2 n} \). Clearly, \((1/2)\log^2 n = o(\sqrt{n} \log \log n) \). Thus, \((\sqrt{n})^{\log n} = o \left((\log n)^{\sqrt{n}}\right)\).

2. Consider the following algorithm:

Pick a random sample \( S \) of \( k \) elements from \( X \); Find and output the minimum element of \( S \). (The value of \( k \) will be fixed in the analysis.)

**Analysis:**

Probability that a randomly picked element of \( X \) has a rank of \( \leq \sqrt{n} \) is \( \geq \frac{1}{\sqrt{n}} \). Thus, the probability that a randomly picked element of \( X \) has a rank \( > \sqrt{n} \) is \( < (1 - \frac{1}{\sqrt{n}}) \). Note that our algorithm will output an incorrect answer only when all the elements in the sample has a rank \( > \sqrt{n} \). Probability of an incorrect answer thus is \( < \left(1 - \frac{1}{\sqrt{n}}\right)^k = \left(1 - \frac{1}{\sqrt{n}}\right)^{\sqrt{n} \log \log n} \leq e^{-k/\sqrt{n}} \). This probability will be \( \leq n^{-a} \) when \( k \geq \alpha \sqrt{n} \log_e n \).

Clearly, the run time of the algorithm is \( O(k) = O(\sqrt{n} \log n) \).

3. Like in the case of Delete-min, we start by replacing \( a[i] \) with \( a[n] \). After this replacement, \( a[n] \) is no longer a part of the heap. We invoke Heapify on the node \( i \). Specifically, we call \( Heapify(a, i, n - 1) \). After this call, the subtree rooted at \( i \) will be heap. Now there are two cases to consider. Case 1: \( a[i/2] < a[i] \). In this case, we are done. Case 2: \( a[i/2] > a[i] \). In this case we swap \( a[i/2] \) and \( a[i] \) and proceed to correct the heap recursively starting from \( a[i/2] \). The total time needed is \( O(\log n) \).

4. We employ a 2-3 tree with the following modification: Each non leaf stores the largest key \( L \) from the first subtree, the largest key \( H \) from the second subtree, and also the number \( S \) of leaves rooted at this node.

**Search\((x)\)** is done like in a standard 2-3 tree.

To process **Find\((i)\):** We start from the root \( r \). Let the \( S \) values stored in the first two children of \( r \) be \( S_1 \) and \( S_2 \). If \( i \leq S_1 \) we move to the first child of \( r \) and proceed
recursively. If \( i > S_1 \) and \( S_1 \leq S_1 + S_2 \) then we move to the second child of \( r \), decrement the value of \( i \) by \( S_1 \), and proceed recursively. If \( i > S_1 + S_2 \) then we move to the third child of \( r \), decrement the value of \( i \) by \( S_1 + S_2 \), and proceed recursively.

When \( i = 1 \) and we are in a leaf, we output the key in this leaf. Clearly, this search takes \( O(\log n) \) time since we only spend \( O(1) \) time at each level of the tree.

To process \( \text{Insert}(x) \), we proceed exactly as in the case of a standard 2-3 tree. The only difference is that we have to update the \( S \) values in addition to the \( L \) and \( H \) values.

The operation \( \text{Delete}(x) \) is also done in a similar way. The time taken for each of these operations is \( O(\log n) \).

5. (a) \( T(n) = 256 T \left( \frac{n}{2} \right) + n^8 \). Here \( a = 256, b = 2, n^{\log_b a} = n^8, f(n) = n^8 \). Case 2 of the Master theorem applies. Therefore, \( T(n) = \Theta(n^8 \log n) \).

(b) \( nT(n) = T(n-1) + T(n-2) + \ldots + T(1) + T(0) \). Also, \( (n-1)T(n-1) = T(n-2) + T(n-3) + \ldots + T(1) + T(0) \). Subtracting the second equality from the first we realize that \( nT(n) - (n-1)T(n-1) = T(n-1) \). I.e., \( nT(n) = nT(n-1) \). This means that \( T(n) = T(n-1) = \cdots = T(1) = T(0) = 1 \).

6. Note that if \( A \) and \( B \) are two sorted sets then we can compute \( A \cap B \) in \( O(|A| + |B|) \) time by merging the two sorted sequences. Given the sets \( A_1, A_2, \ldots, A_m \), start by intersecting \( A_1 \) with \( A_2 \) to get \( B_1 \); \( A_3 \) with \( A_4 \) to get \( B_2 \); \ldots; \( A_{m-1} \) with \( A_m \) to get \( B_{m/2} \). This will take a total time of \( O \left( \sum_{i=1}^{m/2} |A_{2i-1}| + |A_{2i}| \right) \). This in turn is no more than \( cn \) for some constant \( c \). Realize that \( |B_1| \leq \min\{|A_1|, |A_2|\}; |B_2| \leq \min\{|A_3|, |A_4|\}; \) and so on. Next, intersect \( B_1 \) with \( B_2 \) to get \( C_1 \); \( B_3 \) with \( B_4 \) to get \( C_2 \); and so on. These intersections will take time no more than \( c \left( \sum_{i=1}^{m/2} |B_i| \right) \). However, \( \sum_{i=1}^{m/2} |B_i| \leq \frac{n}{2} \). Thus the second stage of intersections will take time no more than \( c \frac{n}{2} \).

Continue the above stages of intersections until we are left with only one set. The time taken in any stage is no more than one half of the time spent in the previous stage. Therefore the run time of the entire algorithm is \( \leq c \left[ n + \frac{n}{2} + \frac{n}{4} + \cdots \right] \leq 2cn = O(n) \).