1. The correct order is
\( n \frac{1000}{\log n} , 2^{\sqrt{2 \log \log n}}, \log n, 0.001n^3, (n + 1)!, 2^{2n + \log n}, n2^n. \)

Here, one can use the trick \( x = 2^{\log x}. \) The first function is a constant since \( n \frac{1000}{\log n} = 2 (\log n) \frac{1000}{\log n} = 2^{1000}. \) Similarly, the function \( \log n \) can be written as \( 2^{\log \log n}, \) which is asymptotically faster than the function \( 2^{\sqrt{2 \log \log n}}. \)

By Stirling’s approximation, \( (n + 1)! = O(n^{n+1}) = O(2^{n \log n}), \) which is slower than the function \( n2^n. \) Finally note that \( 2^{2n + \log n} = 2^n 2^{2n} = n2^n. \)

2. Let us try to analyze the recurrence for the following algorithm for searching in an unsorted array:
check the first element and recurse on the array excluding the first element. Let \( T(n) \) be the worst case running time of this algorithm on an input of size \( n. \) We have
\[
T(n) = T(n - 1) + O(1).
\]

We know that the running time is linear, but we cannot use the master theorem directly, since the subproblem size is not a constant fraction of the input size.

3. Taking out the last term in the summation, we get
\[
T(n) = n + T(n - 1) + \sum_{i=1}^{n-2} T(i)
\]
\[
= 1 + T(n - 1) + \left[ (n - 1) + \sum_{i=1}^{n-2} T(i) \right]
\]
\[
= 2T(n - 1) + 1.
\]

This is now a familiar looking recurrence and can be solved by using the iteration method:
\[
T(n) = 2T(n - 1) + 1
\]
\[
= 2(2T(n - 2) + 1) + 1
\]
\[
= 2^{n-1}T(1) + \sum_{i=0}^{n-2} 2^i
\]
\[
= \sum_{i=0}^{n-1} 2^i = 2^n - 1.
\]
4. One solution is to run the linear time median finding algorithm \( k \) times, that is, once for each interval \( I_i \). This takes \( O(kn) \) time.

A better strategy is to build a data structure so as to reduce the query time for each interval. Recall quicksort with deterministic median finding and partitioning at each step. Suppose we store the partitioned array \( A^i \) at each step \( i \) and store a position lookup array \( Pos^i \) that stores the new position of each element with respect to their original position in \( A \). We require the partitioning procedure to respect the indices in each part, that is, each part in \( Pos^i \) should have sorted indices. This kind of partitioning procedure is easy to implement and still takes \( O(n) \) time. This data structure takes \( O(n \log n) \) time to create and \( O(n \log n) \) space.

Now, given an interval \( I = (l, r) \). We can now find the two parts of the interval \( I \) in \( A^1 \) in \( O(\log n) \) time by doing some binary searches in \( Pos^i \). Thus we know the sizes of the two parts and we can appropriately recurse into the correct part.

One interval query thus takes \( O(\log^2 n) \) time to answer, so the total time complexity is \( O(n \log n + k \log^2 n) \). A faster algorithm can be found in the paper [1].

5. An algorithm like Heapsort [2] can be used for this purpose. It is in-place and runs in \( O(n \log n) \) time. However, one has to take care not to make a recursive call, since we cannot use a non-constant sized recursion stack. It is a fairly straightforward procedure to convert the Heapify procedure in the book into an iterative one.

References
