1. Although the problem didn't require it, here's a Java method that implements the algorithm:

```
static double tmax(double[] A, int low, int high) {
    if (high == low)
        return A[low];
    int mid = (low + high) / 2; // Note:  low <= mid < high
    double x = tmax(A,low,mid);
    double y = tmax(A,mid+1,high);
    return (x > y) ?  x :  y;
}
```

In the recursive case, this method makes two recursive calls to itself, and the problem size in each recursive call is half of the original problem size. The extra work is just computing *mid* and comparing x to y, so the amount of extra work is constant. This leads to the recurrence relation $T(n) = 2 * T(n/2) + \Theta(1)$. Compared to the general form $T(n) = a * T(n/b) + \Theta(f(n))$, we have $a = b = 2$, so $\log_b(a) = 1$, and $f(n) = 1 = n^0$. Since the exponent in $f(n)$ is less than $\log_b(a)$, this is Case 1 of the master theorem. So we conclude that $T(n) = \Theta(n^{\log_b(a)}) = \Theta(n^1) = \Theta(n)$.

2. **a)** The worst case occurs when each of the $\sqrt{N}$ items in the unsorted part of the array has to move all the way down to the start of the array. Each of the $\sqrt{N}$ items moves past between $N - \sqrt{N}$ and N other items, so the worst case run time is $\Omega(\sqrt{N} * (N - \sqrt{N}))$, and it is $\mathcal{O}(\sqrt{N} * N)$. Since $\Omega(\sqrt{N} * (N - \sqrt{N})) = \Omega(\sqrt{N} * N)$, we see that in fact the worst case run time is $\Theta(\sqrt{N} * N) = \Theta(N^{3/2})$.

b) Sort the $\sqrt{N}$ items at the end of the array (using heap sort, for example), which has run time $\Theta(\sqrt{N} * \log(\sqrt{N}))$. Then use the merge operation from Merge Sort to merge the first part of the array (the first $N - \sqrt{N}$ elements) with the second part of the array (the last $\sqrt{N}$ elements). Since a total of N elements are merged, the run time for this part of the procedure is $\Theta(N)$. The total run time to sort the array is then $\Theta(\sqrt{N} * \log(\sqrt{N})) + \Theta(N)$. Since $\sqrt{N} * \log(\sqrt{N})$ is of lower order than N, the combined run time is actually $\Theta(N)$.

3. The algorithm in either case goes like this:

```
Input:  An array of lists list[0]..list[K-1]
Output:  The merged list
mergedList = empty list
while at least one of the given lists is non-empty {
    Find i such that list[i] has the minimal first element
    Remove first element from list[i] and add it to mergedList
}
```

Since there are N elements in all of the lists combined, the while loop runs for N iterations. The difference between the two algorithms is in how to find the i such that $list[i]$ is minimal

a) To find the list with the smallest first element, use the usual algorithm for finding the minimum (comparing two lists by comparing their first elements). If there are K non-empty lists, this requires $K - 1$ comparisons, and the run time is $\Theta(K)$. In the worst case, where all the lists remain non-empty until near the end of the process, the total run time for all N iterations of the while loop is then $\Theta(N * K)$.

b) In the second version of the algorithm, we put the lists into a min-heap of lists, where two lists are compared by comparing their first elements. Building this heap takes time $\Theta(K)$. Then, in each iteration of the while loop, the list with the smallest first element is at the top of the heap. When the first item in that list is removed, the key value for the list changes, so the heap property has to be restored by calling min-heapify() on the root of the heap. This takes time $\Theta(\log(K))$. (When a list becomes empty, it must instead be removed from the heap, as if removeMin() was called.) So, in each iteration of the while loop, finding the list with the minimal first element now takes time $\Theta(\log(K))$ in the worst case, instead of $\Theta(K)$, and the total time for all N iterations of the while loop is $\Theta(K + N * \log(K))$, which is $\Theta(N * \log(K))$.

4. a) If the array is full and we do an insertion, the array size is doubled, with a cost of $\Theta(N)$. If we then immediately delete an item, the array is only half full, and we cut its size in half, with a cost of $\Theta(N)$. If there is a long series of alternating insertions and deletions, we would change the array size for every operation, at a cost of $\Theta(N)$ for each operation. For that sequence of operations, this gives an amortized cost of $\Theta(N)$ per operation instead of $\Theta(1)$ per operation.

b) Instead of dividing the list size by 2 as soon as the list becomes half empty, wait until the list becomes 3/4 empty. That is, when deleting an item, if the number of items in the array drops below 1/4 of the array length, then replace the array with one that is only half as long. No matter what happens next (insertions or deletions), there will have to be a fairly long sequence of operations before the array size has to be changed again. The cost of cutting the array in half is then amortized over the sequence of constant-cost operations that occur before the next size change. For example, if the array size is 1024 and the number of items drops to 256, we change the array's size to 512. There will then be, at a minimum, 256 insertions or 128 deletions before the array size will have to change again (and if there is a mixture of insertions and deletions, then there will be even more cheap operations before the next size change).

5. The cost for algorithm **(a)** is $\Theta(N * \log(N) + M)$, since the cost for sorting N items is $\Theta(N * \log(N))$ and the cost for getting the last M items from the array is just M. The cost for algorithm **(b)** is $\Theta(N + M * \log(N))$, since the cost for building a heap of N elements is $\Theta(N)$, and once the heap is built we have to call removeMax() M times at a cost of $\log(N)$ for each call. The cost of algorithm **(c)** is $\Theta(N + M * \log(M))$: Running the order static algorithm has expected time $\Theta(N)$, and applying the partitioning algorithm also has run time $\Theta(N)$, so those two operations together have run time $\Theta(N)$. Sorting the M elements at the end of the array can be done in time $\Theta(M * \log(M))$. (There is also the task of removing the M elements from the array, but that run time is dominated by $\Theta(M * \log(M))$.)

If M is a constant, then the run times for both algorithms **(b)** and **(c)** reduce to $\Theta(N)$. This is certainly less than the run time for algorithm **(a)**. It's not clear whether **(b)** or (c) is the best choice. If N is large, then $\log(N)$ is also significant, and that might be a reason for preferring (c) to (b). On the other hand, algorithm (c) is more complicated, and that might mean that (b) is actually faster.