Problem 2: Break Quicksort

Answer: [12, 10, 8, 6, 4, 2, 1, 3, 5, 7, 9, 11]

The worst possible case for quicksort (for any pivot) would be if we chose the pivot as the largest variable the array in every iteration—this would result in having to *call* quicksort $\theta(n)$ times because only the largest element would be sorted every time and each quicksort call would be $\theta(n)$ complexity because it would still have to go through all elements of each array in the forloop giving us $\theta(n^2)$. However, we cannot do the absolute worst case, or the largest element in every array/subarray because by definition of median, the largest element of each subarray can never be picked; therefore to achieve the worst case we must do the second to worst thing and make sure that the median/pivot is the second largest element every time—for the first, initial call it must be the second largest 11, so we put 11 on the rightmost (R) spot and 12 on the leftmost (L) spot and since 11 and 12 are the largest two, the middle element for medianof3() function will be less than 11 and 12 thereby making 11 the pivot as we intended. By this logic of only choosing the second largest element as pivot/median for the worst case scenario (going off our model where elements range from 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12) we should have it such that a[L] is always the second largest element and the largest (even) element will be at a[R] so that the function sorts only those two largest elements and effectively pops them from the array (it's still stored obviously, but in terms of what is left to sort, it is popped). Because it is effectively the first and last elements that are "popped", we want to make sure that 12 is on the left hand side followed by 10, 8, 6, 4, 2 (evens in descending order going right) and that 11 is on the right hand side 1, 3, 5, 7, 9, 11 (odds in descending order going left), giving us our answer [12, 10, 8, 6, 4, 2, 1, 3, 5, 7, 9, 11]. This will be worse case—or "popping" just the (even) largest element and the (odd) second largest element—because when observing the code, because the a[L] will always be our largest (even) element and a[R] will be our second largest (odd) element, a[(L+R)/2] will be less than both and thus index of the median, index would always equal R, making a.swap(index, R) not change anything. Then p will be set to a[R], which we established is the second largest element. The forloop will then effectively do bubble sort on the largest element at a[L] and push it all the way up until a[R-1] and making i be R-1 as well. We then would swap a[R-1] and a[R], putting the largest element at a[R] and the second largest at a[R-1], and since these two are sorted, partition returns i = R-1—effectively "popping" the last two elements from the array, giving the subarray to work with for the next recursive call for Quicksort(a, L, partition(a,L,R) - 1). We know that Quicksort(a, L, partition(a,L,R) + 1) will just run its course as well, but because we engineered the left hand side of the pivot to always be the worst case in this scenario (where LHS of pivot is array.size()-2 and RHS of pivot is 1), runtime for the other partition will be significantly more. After we "pop" the last two elements of the subarray, we would similarly be left with the new largest element occupying new index L and the new second largest occupying index R (10 and 9 in the 2nd recursive call), which we know will be the worst case possible almost inductively by the last step. This would mean that quicksort will be called n/2 times because each call 2 elements get sorted, and since partition is $\theta(n)$, the worst case would still be $\theta(n^2)$.