1) Describe or write (psudeo)code to automatically generate tests for this function [myqsort(int a[], unsigned int size)].

To generate tests for this function, one simple solution is to blitz the function with a series of random tests. These random tests consist of generating random input to the function, calling the function with the random input, then checking to see if the sorting function has correctly sorted the data. A single such test case will broadly consist of the following steps, assuming all overhead (seeding psuedorandom number generators, etc.) has been completed:

1. Pick a psuedorandom size of an array to fill, and declare/allocate memory for that array.
2. Fill each element of that array with another psuedorandom number (which may be either the same as its predecessor or not). Additionally, a stronger testing strategy would be to include "empty elements" in the array in order to screw with comparison statements in myqsort [aside: these test cases should probably fail, so looking for failing behavior with empty/semiempty arrays is a good strategy].
3. When the array is filled, either
   A) pass its size to myqsort directly; **or**
   B) choose another psuedorandom number that is **not** the size of the above allocated array and pass it to myqsort just to screw with it (again these should probably fail, so looking to see that associated tests on the result of myqsort fail is an important step). This is probably an operation to perform fewer times than the alternative option, as it only tests to see if myqsort() correctly identifies a discrepancy in the size of a and the value you give it.
4. After the array has been constructed and its size (or another random variable not its size) have been filled/determined, call myqsort() with the determined arguments and watch it burn (or not, as the case may be). Again note that with purposely erroneous input, the function is expected to not fail, so a passing set of tests on bad input is not a positive sign (see input domain partitioning; these bad tests can be separated into their own subdomain and partitioned further from there if needed).
5. And finally examine the results of your inputs, as seen below.

2. Describe or write (psudeo)code to see if results are correct; in particular, is following check sufficient? If not, how could it be improved?

One way to check test cases with non-bad input (i.e. not empty arrays, not wrong sizes, etc.) is to begin with the first element in the sorted array and iterate through the remaining elements, checking that the examined element is indeed less than or equal to the elements that come after it. Then move on to the second element, and perform the same process (being sure to exclude already examined elements).

Continue this process until the last element is reached, which theoretically is the greatest element in the array (though for the paranoid, checking that all other elements are less than/equal to it is a good idea). For the truly paranoid, you can start with the last element, check it is greater than or equal to all other elements, then repeat with the next to last element, then the element before that, etc. ad nauseum.

Finally, ensure that the elements of the original array are still there in correct quantities in order to ensure myqsort() hasn't screwed with the contents of the array you passed it in any capacity except to sort it. Test cases with purposefully bad input are simpler; check to see that the function fails the comparison checks given above in at least one place.

Essentially, these test cases are a huge series of comparison operators and checks to ensure that the goal of the function myqsort() has been accomplished: the array given as input is sorted after it has been passed to the function correctly.

The check given is fairly solid but not complete. The check to iteratively compare the elements of a to its neighbors is correct, but the check of the contents of a is not satisfactory to me. For example, say that a = [0,4,2,3,3,3,3,3,3,1]. If myqsort operates and afterward the array is [0,1,2,3,3,3,3,3,3,4] the test passes, but if the array is [0,1,2,3,4] the given test also passes even though a portion of the array has been lost. Adding a check to ensure that the size of the array after the function has been called, or adding an element-wise comparison check (where each element of a0 must be present in a **unique** element of a) will get around this error.

3. Finally, how would you check that the test generation and check for correctness are sufficiently strong to test quick sorting?

One semi-arbitrary way to check this is via coverage, of which there are four central types. The first, graph testing, makes an analogy of the flow of the function to a mathematical graph as a set of nodes/edges, then asks which nodes, edges (or optionally paths) have been covered by your test case. This is an optimal test coverage method if the source code is known, and allows you to ensure your test cases have at least touched on each portion of the code. However, it does not allow you to determine how rigorously you have tested each node/edge/path of the code, only that it is somehow involved in the test case, so care should be taken to determine **how** each node/edge is used.

One tool to ensure this is the second main type of coverage, logic coverage. Logic coverage is a method that involves replacing logical statements in code by more complicated ones that require more things to be true/not true in the program state and then examining the success/behavior of the program with these new requirements. It allows us to ensure the program has completed the correct behavior for the correct logical reasons.

Input space/domain partitioning allows the different types of inputs to the program to be defined in different “pariwise disjoint” (non-overlapping) sections, then selects one input from each type of section to give the program for a series of tests. For example, assume it is known that (perhaps from static analysis of source code) testing two arrays of zeros, one of size 2 and one of size 200 will produce the same behavior of the function. These test cases are part of the same input domain and thus only one need be run for comprehensive testing.

Finally, syntax-based testing or mutation testing is a fourth way to examine the strengths of the test suite we have created. It generates “mutants”, versions of the program with changes in their structure, and determines how many mutants the test suite “kills”; basically, how much could myqsort() change its behavior and your test suite catch these changes in behavior (assuming these changes led to undesirable behavior).

Between these four methods (and perhaps asking a friend, colleague or simply looking over your test cases manually in a form of static analysis), you should have pretty good confidence that your test suite is effective, or be alerted that your test suite is indeed not effective and needs refining.