Here are a few notes on mergesort.

Merge Procedure

Here is a description of the merge procedure in pseudo-code, which seeks to merge two sorted (assume that they are in ascending order) arrays $L_1$ and $L_2$ (of size $n_1$ and $n_2$ respectively) and produce a sorted array $L$ (of size $n_1 + n_2$) (in ascending order).

$$L = \text{Merge} \ (L_1, L_2, n_1, n_2)$$

BEGIN

1. $p_1 = 1$ [pointer to the current position in the first array]
2. $p_2 = 1$ [pointer to the current position in the second array]
3. while ($p_1 \leq n_1$ AND $p_2 \leq n_2$)
   
   (3a) if ($L_1(p_1) \leq L_2(p_2)$), then append $L_1(p_1)$ to $L$ and increment $p_1$,
   
   (3b) else append $L_2(p_2)$ to $L$ and increment $p_2$.
   
   } [end while loop]
4. if $p_1 > n_1$ but $p_2 < n_2$, then append the remaining elements of $L_2$ (i.e. all elements $L_2(p_2)$ to $L_2(n_2)$) onto $L$.
5. elseif $p_2 > n_2$ but $p_1 < n_1$, then append the remaining elements of $L_1$ (i.e. all elements $L_1(p_1)$ to $L_1(n_1)$) onto $L$.

END

Time Complexity:
The time complexity of this operation is $O(n_1 + n_2)$. Why? Look at the while loop. It will exit when $p_1 > n_1$ or when $p_2 > n_2$ or both. In the former case, we see that all $n_1$ elements of $L_1$ must surely have been copied into $L$, which takes a total of $n_1$ operations. Now let us assume that $x$ elements of $L_2$ got copied into $L$ ($x$ could be 0). This takes another $x$ operations. Once we are out of the while loop, there are still $n_2 - x$ elements remaining in $L_2$ and these need to be copied into $L$, which takes $n_2 - x$ more operations. Hence the total number of operations is $n_1 + n_2 - x + x = n_1 + n_2$.

The textbook mentions that in the worst case, the number of comparisons taking place in the above routine is at most $n_1 + n_2 - 1$. Let us look at this, with an example on two sorted arrays $L_1 = \{1, 3, 5\}$ and $L_2 = \{2, 4, 6\}$.

We initialize $p_1 = p_2 = 1$. As we have $L_1(1) < L_2(2)$, we append $L_1(p_1 = 1)$ to $L$. So $L$ now contains the element 1, and $p_1$ is incremented to 2. The next time, we see that $L_2(p_2 = 1)$ is less than $L_1(p_1 = 2)$ and hence we have $L = \{1, 2\}$. Now $p_2$ is incremented to 2. In the next step, we have $L_1(p_1) < L_2(p_2)$ and hence $L = \{1, 2, 3\}$. If you continue likewise you will see that at some stage, we have $p_1 = 3$ and $p_2 = 3$ and $L = \{1, 2, 3, 4\}$. Now the element 5 from $L_1$ gets added to $L$, and $p_1$ is incremented to 4. As $p_1 > n_1$ the while loop exits. So far, we have performed a total of 5 comparison operations and 5 copying operations. We need to perform one more copying operation, which is putting the number 6 from $L_2$ into $L$.

In a general case, the total number of comparison operations will be $n_1 + n_2 - 1$ and the total number of copying operations will be $n_1 + n_2$. The total of both comparison and copying operations will be less than $2(n_1 + n_2)$. In any event, the time complexity of the merge procedure is $O(n_1 + n_2)$.
Merge Sorting

Look at the pseudo-code for mergesort on page 318 of the book. It recursively divides the array in a peculiar way until you get just individual elements. The merge procedure is then applied bottom-up. In class, I simplified this by asking you to directly look at individual elements and merge them to produce sorted sub-arrays of size 2 each, then take adjacent sub-arrays of size 2 and merge them to produce sorted sub-arrays of size 4, and so on, until you get one final sorted array. In class, we derived the time complexity of the entire mergesort procedure to be $O(n \log n)$. Here is a gist: There are some $k$ steps, in each of which we spend $O(n)$ time in the different merging operations. What is the maximum value of $k$? Again: In the first step, we had $n$ arrays each of size 1 which were merged to give you $n/2$ arrays each of size 2, which again were merged in the second step to give you $n/4$ arrays of size 4. This will go on for $k$ steps. In the $k^{th}$ step, a total of $n/2^{k-1}$ arrays, each of size $2^{k-1}$ are merged to give the final array. Clearly this can go on only until $2^k = n$, i.e. until $k = \log n$. Hence the total number of steps is equal to $\log n$, yielding an overall time complexity of $O(n \log n)$.

I assumed, in this case, that the size of the original array was in the form $n = 2^m$ where $m$ is an integer. The natural question is what if $n$ is not an integral power of two. There are several things you can do. Here is one: Let us suppose that $n$ wasn’t an integral power of 2, so that $2^m < n < 2^{m+1}$. Now let us add some $x$ dummy elements to the array (a dummy element could be something like a very large number) so that $n + x = 2^{m+1}$. In the very worst case, we need to add $x = 2^m - 1$ dummy elements and this will at most double the size of the original array. In other words, we now would have $n' = 2^{m+1}$ elements instead of $n$ elements (and note that $n' < 2n$). Now apply the mergesort procedure to this new array. The total time complexity is $O(n' \log n')$, i.e. $O(2n \log 2n)$, which is no worse than $O(n \log n)$. There are many other tricks some of which will be quicker by a constant factor, but they will not affect the time complexity. Therefore we lose nothing by restricting ourselves to arrays whose size is an integral power of 2 (if at all, it simplifies the analysis).