Example input:

2	4	8	3	7	1	5	6

Mergesort

Recursively mergesort each half of the array, and then combine the two halves into a sorted array. This combining is done by looking at the first item of each array, and inserting the smaller of the two as the next item in the sorted array, until one of the arrays is exhausted, and then the remaining elements of the other array can be inserted directly.

The amount of time to combine two sorted arrays of size $N/2$ into a fully sorted array takes time N, since each comparison only removes one item from one of the arrays. A recurrence relation can be used to show the running time is $O(N \log N)$. Let $T(N)$ be the time it takes mergesort to sort an array of length N. Then we have that

$$\begin{aligned}
T(N) &= N + 2T(N/2) \\
&= N + 2(N/2 + 2T(N/2^2)) \\
&= N + N + 2^2T(N/2^2) \\
&= 2N + 2^2(N/2^2 + 2T(N/2^3)) \\
&= 3N + 2^3T(N/3^3) \\
&... \\
&= N \log N + 2^{\log N}(T(N/2^{\log N})) \\
&= N \log N + 2^{\log N} \cdot 1 \\
&= N \log N + N \\
&= N(\log N + 1) = O(N \log N)
\end{aligned}$$

This can also be seen using a tree, where each node stores the number of comparisons used in that call. Mergesort needs an extra array of size N to store the results, i.e., it cannot be done in place.

Quicksort

Choose a pivot from the unsorted array. For now, choose the leftmost, but we will talk more about this later. Have two pointers, one to the beginning of the array and one to the end. Compare the beginning item with the pivot, and if it is smaller, increment the beginning pivot, and if it is larger, switch it with the item at the end pointer and decrement that pointer. Continue on this way until the beginning and end pointers match. Then switch the pivot with the item at the pointer, and recursively call quicksort on the array to the left of the pointer and the array to the right.

Pivot Selection

Here are some possible pivot selection strategies:

Leftmost/Rightmost: Choose the leftmost or rightmost element in the array as the pivot.

Random: Choose an element from the array uniformly at random and use that as the pivot, i.e., choose a random number between 1 and N, and whatever element is at that index in the array is the pivot.

Median of Three: Choose the leftmost, rightmost, and middle element in the array and choose the pivot as the median of those three.

Pivot selection matters! For example, if your strategy is leftmost but your array is in reverse sorted order, the running time is N^2 because each call to quicksort splits the array into a group of size 1 and a group of size $N - 1$, instead of splitting it into two groups of equal size.

Heapsort

The basic idea of heapsort is simple... create a heap of the items, and then remove items from the heap until it is empty, and you will remove them in (possibly reverse) sorted order.

Recall that a heap is a priority queue that is a binary where each node is larger than its children. As a review, try building a heap out of the letters of the word "example". Inserting involves adding elements to the end and then "swimming" them up. Deleting involves removing the top element, replacing it with the last element, and "sinking" that element down. Note that when we do this, the left and right trees are heaps so the sink maintains the heap. Both creating a heap this way and removing all elements take time $O(N \log N)$, because you put in (or remove) N items and each item has to go up (or the new root has to go down) the tree a maximum of $\log N$ times before we are done.

Recall also that a heap can be stored in an array, where the children for a node at index i reside at $2(i+1)-1$ and $2(i+1)$. So "example" in an array could be:

x	m	p	l	e	e	a

Now for the sort. We actually want to build the heap so that the top of the heap to be the largest element, and we'll see why soon. We could build the heap in an obvious way, but there is actually a better way. We can build the heap bottom up by pretending we have larger and larger heaps. The first item we check is roughly at $N/2$, and we verify that the heap there is correct, sinking the root if it is not. Then we move up one element and do the same. We continue until we get to the first element, at which point the entire array is the heap. Why is this faster? Well, $N/2$ might have to sink down at least one place, $N/4$ might have to sink down an additional 1 space, $N/8$ might have to sink an additional space on top of that... so we get at most $N/2 + N/4 + N/8 + ... + 1 < N$ sinks, and at most twice that many comparisons.

After the array is a heap, making it into a sorted array is easy. Simply remove the top item with the last item in the unsorted part of the array, and sink the new root to restore the heap, and continue doing this until the heap is empty.