In our ever increasingly digital age, the ability to sort data accurately and efficiently plays a critical role to information systems. Indeed, it is difficult to imagine a world without data being placed in some logical sequence. Consider trying to find a single book in a library. If that library had one miles-long bookshelf and stored books in no particular order, such a task might prove impossible. To be able to find specific data is just as important as the ability to store said data. In fact, an argument could be made that unsorted data is just as meaningless as no data at all.

It follows that the question that must be answered is not “should this collection of data be sorted?”, but rather “what is the best way to sort data?” There are many considerations to be made when deciding which sorting algorithm to employ. Is extra memory required? Is the sort stable? How difficult is the algorithm to implement? The answers to these questions will depend on the unique circumstances of every different program. However, in most cases, the speed of a sorting algorithm will be of primary concern. Presented here is an algorithm, Mergesort, which guarantees very fast sorting, even in the worst case.

Mergesort relies on the power of recursion to divide an array of data into smaller and smaller subarrays. It then builds the array back up by merging the subarrays into sorted order. By sorting in this fashion, Mergesort guarantees a worst-case runtime faster than both Insertion Sort and Quicksort. In addition, Mergesort’s speed is not dependent on the data that it is being used to sort. Insertion Sort’s speed relies on a data set already partially sorted and Quicksort needs its data set to be randomly distributed to perform in the best case. Furthermore, because of its use of recursion, Mergesort requires relatively little code to implement, making it an easy addition to any program.

The first step to implementing Mergesort is to have an array of unsorted data and a second, auxiliary array of the same size. The auxiliary array is used to store values while merging the smaller, subarrays back together. Without this secondary array, keeping track of swapped values would become very complicated. The extra space required for Mergesort is its primary drawback, but as memory is always becoming more abundant, this weakness will continue to diminish in importance.

There are two distinct methods employed by Mergesort - the sort() method and the merge() method. The sort() method is the recursive part of Mergesort and works by dividing the array in half with each call. It is important to note that sort() does not make any changes to the original data in the array. The subarrays being considered are simply portions of the original array, bound by tracking indices. Here those indices will be called Low, Mid and High for the indices of the first item, middle item and last item in the array, respectively.

The sort() method, being recursive, must contain a base case to break the recursive cycle. The base case in this instance is when the subarray being considered contains only one value. Logic and common sense show that an array containing a single value is already sorted. The base case occurs when the subarray has a High index that is less than or equal to its Low index.

Once it is determined that the subarray still requires further deconstruction (that the subarray contains more than one element), a middle point of the current subarray must be calculated. Mid will serve as the upper bound of the left subarray, while the element directly following Mid will serve as the lower bound of the right subarray. So, after calculating Mid, sort() makes two recursive calls to itself: one on the subarray from Low to Mid, and one on the subarray Mid+1 to High. It is with these calls that sort() breaks the array into two halves. Each subsequent call to sort will break its subarray into two more halves, and so on in this way until the base case is reached.

Once all calls to the sort() method have been reduced to the base case, the merge() method is called from sort() to merge the two current subarrays together into a larger, sorted subarray. First, the elements of the current subarrays are copied into the auxiliary array in their current order. Then, a loop is used to examine and compare the elements to determine which should be placed in the merged subarray first. There are four possibilities that need to be considered. If the left subarray is empty, copy the element from the right subarray. If the right subarray is empty, copy the element from the left subarray. If the element being examined in the right subarray is less than the element in the left, copy the element from the right subarray. And finally, if the element being examined in the right subarray is greater than or equal to the element in the left, copy the element from the left subarray.

Mergesort can be implemented in this way for any type of data, provided that there is enough memory for the auxiliary array and that the data can be compared using less than or greater than operators. The above approach is what is called the Top-Down Mergesort. The recursive call stack would lead to the first and second elements being merged into a two element subarray, then the third and fourth elements into another two element subarray, before finally merging those two subarrays into a four element subarray. There is a Bottom-Up approach to Mergesort that sorts every two element subarray first, then creates all four element subarrays, and so on. Instead of recursion, the Bottom-Up approach uses two loops to cycle through the subarrays.

When considering the benefits of using Mergesort, it is important to examine other sorting algorithms and, by comparison, determine why Mergesort should be preferred. Insertion Sort and Quicksort, as touched upon earlier, are two such algorithms that have proven effective in regards to solving the problem of sorting.

The Insertion Sort algorithm works by considering each element in the array and moving it as far to the front as it can. It does so by comparing it to the previous element, exchanging places with that element should it need to, then comparing it to the new previous element. Each value is moved toward the front of the array in this fashion until it is at its current correct position. It is important to note that while being sorted, the values to the left of the element currently being sorted are in order, but they are not necessarily in their final positions. Also, Insertion Sort does its sorting in place - in other words, an extra copy of the array is not needed.

While Insertion Sort is capable of fast runtimes, analysis shows that the speed with which it completes its task is heavily dependent on the data being sorted. If the array is mostly in order before the sort begins, Insertion Sort will need to make relatively few compares and exchanges, leading to a very fast runtime. Insertion Sort is also stable, meaning that the relative order of equal elements is maintained during the sort. However, if the array is in random order prior to the sort, Insertion Sort slows to a crawl. Realistically, the average case for Insertion Sort leads to quadratic runtimes, O(n2), where n is the number of data elements to be sorted. This makes Insertion Sort unrealistic as a solution to sorting large amounts of data.

Quicksort, like Mergesort, uses recursion to divide the array into pieces to allow for faster sorting. With Quicksort, a partition value is selected and the array is partially sorted such that every value less than the partition is to the left, and every value greater than the partition is to the right. This is accomplished by simultaneously moving two indices from either end of the array inward to find the first element on the left that is greater than the partition and the first element on the right that is less than the partition. Then these two values are swapped. This process continues until the partition value is in its final place in the array and is surrounded by two subarrays – the left subarray with smaller values and the right subarray with larger values. These swaps occur in place, eliminating the need for additional storage. Quicksort is then recursively called on each subarray until the entire array is sorted.

Analysis of Quicksort shows that, like Insertion Sort, its speed is dependent on the data being sorted. In this case, the selection of the correct partition point can make or break Quicksort’s effectiveness. To combat the worst case, it often makes sense to randomly shuffle the data being operated on beforehand, ensuring that the worst possible partition points are not selected each time. In realistic cases (given a set of randomly ordered data), Quicksort is very efficient and demonstrates a logarithmic runtime, O(n log n). However, while it may be unlikely, there exists the possibility of a quadratic runtime. This would occur when the smallest remaining unsorted value was selected as each partition point. Another downside to Quicksort is that it is unstable, making it infeasible for sorting jobs that require data stability.

If Quicksort offers the same realistic runtimes, why use Mergesort? As can be seen, the primary advantage that Mergesort offers is that it is completely independent of the data being sorted. Mergesort offers a logarithmic runtime, O(n log n), in the absolute worst case. This makes Mergesort ideal for sorting large sets of data that are of unknown order. Whether the data is completely random or is already in some kind of order, Mergesort will perform quickly. It is the ideal sort to use when the possibilities of quadratic runtimes, no matter how remote, are unacceptable. Mergesort is also stable, making it more versatile than Quicksort.

Even with its advantages, Mergesort does suffer some shortcomings. As touched upon earlier, Mergesort does not do its sorting in place. As such, it requires additional memory equal to what is needed to store the data being sorted. Other sorting algorithms would be better solutions for situations where memory conservation is important. Mergesort is also impractical for small sets of data. Being of recursive design, the overhead of the method calls makes Mergesort relatively slow for these small data sets.

Looking forward, Mergesort should prove itself extremely useful in the field of computer science, but strides should always be made to improve the algorithm. Due to its lack of speed on small arrays, perhaps using Insertion Sort on the smaller sets of data would speed up the runtime. This modification would allow Mergesort to do the heavy lifting by dividing up the data. Then, once a small enough subarray is being considered, Insertion Sort could be implemented on said subarray. While Mergesort cannot operate without the extra required memory, steps can be taken to reduce the amount of time required to copy data to the extra array. Also, by adding simple tests to determine if the array (or subarray) is already in order, we can reduce the time spent on those arrays.

Mergesort, like the other sorts discussed here, still relies on comparing values. As such, it cannot possibly perform with runtimes better than the logarithmic. There is more work is to be done on the problem of sorting and there are new algorithms to be discovered. Ideally, there would exist a sorting algorithm that did not rely on comparing values and that also sorted stably, in place, and with a logarithmic or better runtime. Only continually striving for improvement will determine the future of sorting.