The ability to sort data accurately and efficiently plays a critical role to information systems in our ever increasingly digital age. Indeed, it is difficult to imagine a world without data being placed some logical sequence. Consider trying to find a single book in a library if that library had one miles-long shelf that stored books in no semblance of 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 are 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 fast sorting in the worst case.

Mergesort relies on the power of recursion to divide an array of data into smaller and smaller subarrays, and then builds the array back up by merging the subarrays in sorted order. By sorting in this fashion, Mergesort guarantees a worst-case runtime faster than that of Insertion Sort, or even that of Quicksort. Mergesort’s speed does not rely 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. In addition, 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 very complicated. The extra space required for Mergesort is its primary drawback, but memory is always becoming more abundant, thus diminishing Mergesort’s weakness.

There are two distinct methods employed by Mergesort. The sort() method and the merge() method. The sort() method is the recursive part of Mergesort, working to halve the array 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 by nature 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 it contains more than one element in other words, 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 their base cases, the merge() method is called from sort() to merge the two current subarrays together into a larger, sorted subarray. It does so by first copying the elements of the current subarrays into the auxiliary array in 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 that can be compared using less than or greater than operators, provided that there exists enough memory to account for the auxiliary array. 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 and then the third and fourth elements into another two element subarray before merging those two subarrays into a four element subarray. There is a Bottom-Up approach to merge sort that sorts every two element subarray first, then creates all four element subarrays and so on. This Bottom-Up approach uses two loops, instead of the recursive calls, 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 touch 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 in this fashion down the array until it is at its current place in the array. Insertion Sort does its sorting in place. In other words, an extra copy of the array is not needed. While being sorted, the values to the left of the element currently being sorted are in order, by they are not necessarily in their final positions.

Analysis of Insertion Sort shows that the speed with which it completes its tasks is heavily dependent on the data being sorted. Should the array be mostly in order before the sort, Insertion Sort will need to make relatively few compares and exchanges and will finish very quickly indeed. Insertion Sort is also stable, meaning that the relative order of equal elements is maintained during the sort. However, if the array is anywhere near 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, similarly to 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, finding the first element on the left that is greater than the partition, finding the first element on the right that is less than the partition, and swapping these two values. These swaps continue 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, which eliminates any 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 to ensure that the worst possible partition points aren’t selected each time. In the realistic case, given a set of randomly ordered data, Quicksort is very efficient, with a logarithmic runtime, O(n log n). However, while it may be unlikely, there exists the possibility of quadratic runtime. This would occur when the smallest remaining unsorted value was selected as each partition point. Quicksort is also an unstable sort, making it infeasible for sorting requiring 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 order of the data being sorted. In the worst case, Mergesort offers a logarithmic runtime, O(n log n). 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 a logical sorted order, Mergesort will perform quickly. It is the ideal sort to use when the possibilities of quadratic runtimes, no matter how remote, are unacceptable. Also, Mergesort, like Insertion Sort, is stable.

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 may be better in situations where memory is an issue. 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 incorporating Insertion Sort into the smaller sets of data would speed up the runtime. Mergesort would 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 can’t operate without the extra memory required, steps can be taken to reduce the amount of time required to copy data to the extra array. Also, by adding test to determine if the array (or subarray) is already in order, we can reduce the time taken on those arrays.

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