**Quick Sort**

The **quick sort** uses divide and conquer to gain the same advantages as the merge sort, while not using additional storage. As a trade-off, however, it is possible that the list may not be divided in half. When this happens, we will see that performance is diminished.

A quick sort first selects a value, which is called the **pivot value**. Although there are many different ways to choose the pivot value, we will simply use the first item in the list. The role of the pivot value is to assist with splitting the list. The actual position where the pivot value belongs in the final sorted list, commonly called the **split point**, will be used to divide the list for subsequent calls to the quick sort.

[Figure 12](https://runestone.academy/runestone/books/published/pythonds/SortSearch/TheQuickSort.html#fig-splitvalue) shows that 54 will serve as our first pivot value. Since we have looked at this example a few times already, we know that 54 will eventually end up in the position currently holding 31. The **partition** process will happen next. It will find the split point and at the same time move other items to the appropriate side of the list, either less than or greater than the pivot value.

Diagram

Description automatically generated

Figure 12: The First Pivot Value for a Quick Sort

Partitioning begins by locating two position markers—let’s call them leftmark and rightmark—at the beginning and end of the remaining items in the list (positions 1 and 8 in [Figure 13](https://runestone.academy/runestone/books/published/pythonds/SortSearch/TheQuickSort.html#fig-partitiona)). The goal of the partition process is to move items that are on the wrong side with respect to the pivot value while also converging on the split point. [Figure 13](https://runestone.academy/runestone/books/published/pythonds/SortSearch/TheQuickSort.html#fig-partitiona) shows this process as we locate the position of 54.

Diagram

Description automatically generated

Figure 13: Finding the Split Point for 54

We begin by incrementing leftmark until we locate a value that is greater than the pivot value. We then decrement rightmark until we find a value that is less than the pivot value. At this point we have discovered two items that are out of place with respect to the eventual split point. For our example, this occurs at 93 and 20. Now we can exchange these two items and then repeat the process again.

At the point where rightmark becomes less than leftmark, we stop. The position of rightmark is now the split point. The pivot value can be exchanged with the contents of the split point and the pivot value is now in place ([Figure 14](https://runestone.academy/runestone/books/published/pythonds/SortSearch/TheQuickSort.html#fig-partitionb)). In addition, all the items to the left of the split point are less than the pivot value, and all the items to the right of the split point are greater than the pivot value. The list can now be divided at the split point and the quick sort can be invoked recursively on the two halves.

Diagram

Description automatically generated

Figure 14: Completing the Partition Process to Find the Split Point for 54

The quickSort function shown in [ActiveCode 1](https://runestone.academy/runestone/books/published/pythonds/SortSearch/TheQuickSort.html" \l "lst-quick) invokes a recursive function, quickSortHelper. quickSortHelper begins with the same base case as the merge sort. If the length of the list is less than or equal to one, it is already sorted. If it is greater, then it can be partitioned and recursively sorted. The partition function implements the process described earlier.