Sorting

By **timmac** — *TopCoder Member*
Discuss this article in the forums

Introduction

Any number of practical applications in computing require things to be in order. Even before we start computing, the importance of sorting is drilled into us. From group pictures that require the tallest people to stand in the back, to the highest grossing salesman getting the largest Christmas bonus, the need to put things smallest to largest or first to last cannot be underestimated.

When we query a database, and append an ORDER BY clause, we are sorting. When we look for an entry in the phone book, we are dealing with a list that has already been sorted. (And imagine if it weren't!) If you need to search through an array efficiently using a binary search, it is necessary to first sort the array. When a problem statement dictates that in the case of a tie we should return the lexicographically first result, well… you get the idea.

General Considerations

Imagine taking a group of people, giving them each a deck of cards that has been shuffled, and requesting that they sort the cards in ascending rank order. Some people might start making piles, others might spread the cards all over a table, and still others might juggle the cards around in their hands. For some, the exercise might take a matter of seconds, for others several minutes or longer. Some might end up with a deck of cards where spades always appear before hearts, in other cases it might be less organized. Fundamentally, these are all the big bullet points that lead algorithmists to debate the pros and cons of various sorting algorithms.

When comparing various sorting algorithms, there are several things to consider. The first is usually runtime. When dealing with increasingly large sets of data, inefficient sorting algorithms can become too slow for practical use within an application.

A second consideration is memory space. Faster algorithms that require recursive calls typically involve creating copies of the data to be sorted. In some environments where memory space may be at a premium (such as an embedded system) certain algorithms may be impractical. In other cases, it may be possible to modify the algorithm to work "in place", without creating copies of the data. However, this modification may also come at the cost of some of the performance advantage.

A third consideration is stability. Stability, simply defined, is what happens to elements that are comparatively the same. In a stable sort, those elements whose comparison key is the same will remain in the same relative order after sorting as they were before sorting. In an unstable sort, no guarantee is made as to the relative output order of those elements whose sort key is the same.

Bubble Sort

One of the first sorting algorithms that is taught to students is bubble sort. While it is not fast enough in practice for all but the smallest data sets, it does serve the purpose of showing how a sorting algorithm works. Typically, it looks something like this:

```
for (int i = 0; i < data.Length; i++)
   for (int j = 0; j < data.Length - 1; j++)
      if (data[j] > data[j + 1])
      {
         tmp = data[j];
         data[j] = data[j + 1];
         data[j + 1] = tmp;
      }
```

The idea is to pass through the data from one end to the other, and swap two adjacent elements whenever the first is greater than the last. Thus, the smallest elements will “bubble” to the surface. This is $O(n^2)$ runtime, and hence is very slow for large data sets. The single best advantage of a bubble sort, however, is that it is very simple to understand and code from memory. Additionally, it is a stable sort that requires no additional memory, since all swaps are made in place.

Insertion Sort

Insertion sort is an algorithm that seeks to sort a list one element at a time. With each iteration, it takes the next element waiting to be sorted, and adds it, in proper location, to those elements that have already been sorted.

```
for (int i = 0; i <= data.Length; i++) {
   int j = i;
   while (j > 0 && data[i] < data[j - 1])
      j--;
   int tmp = data[i];
   for (int k = i; k > j; k--)
      data[k] = data[k - 1];
   data[j] = tmp;
}
```

The data, as it is processed on each run of the outer loop, might look like this:

```
{18,  6,  9,  1,  4, 15, 12,  5,  6,  7, 11}
{ 6, 18,  9,  1,  4, 15, 12,  5,  6,  7, 11}
{ 6,  9, 18,  1,  4, 15, 12,  5,  6,  7, 11}
{ 1,  6,  9, 18,  4, 15, 12,  5,  6,  7, 11}
{ 1,  4,  6,  9, 18, 15, 12,  5,  6,  7, 11}
{ 1,  4,  6,  9, 15, 18, 12,  5,  6,  7, 11}
{ 1,  4,  6,  9, 12, 15, 18,  5,  6,  7, 11}
{ 1,  4,  5,  6,  9, 12, 15, 18,  6,  7, 11}
{ 1,  4,  5,  6,  6,  9, 12, 15, 18,  7, 11}
{ 1,  4,  5,  6,  6,  7,  9, 12, 15, 18, 11}
{ 1,  4,  5,  6,  6,  7,  9, 11, 12, 15, 18}
```

One of the principal advantages of the insertion sort is that it works very efficiently for lists that are nearly sorted initially. Furthermore, it can also work on data sets that are constantly being added to. For instance, if one wanted to maintain a sorted list of the highest scores achieved in a game, an insertion sort would work well, since new elements would be added to the data as the game was played.

Merge Sort

A merge sort works recursively. First it divides a data set in half, and sorts each half separately. Next, the first elements from each of the two lists are compared. The lesser element is then removed from its list and added to the final result list.

```
int[] mergeSort (int[] data) {
   if (data.Length == 1)
      return data;
   int middle = data.Length / 2;
   int[] left = mergeSort(subArray(data, 0, middle - 1));
   int[] right = mergeSort(subArray(data, middle, data.Length - 1));
   int[] result = new int[data.Length];
   int dPtr = 0;
   int lPtr = 0;
   int rPtr = 0;
   while (dPtr < data.Length) {
      if (lPtr == left.Length) {
         result[dPtr] = right[rPtr];
         rPtr++;
      } else if (rPtr == right.Length) {
```

```
            result[dPtr] = left[lPtr];
            lPtr++;
         } else if (left[lPtr] < right[rPtr]) {
            result[dPtr] = left[lPtr];
            lPtr++;
         } else {
            result[dPtr] = right[rPtr];
            rPtr++;
         }
         dPtr++;
      }
      return result;
   }
```

Each recursive call has O(*n*) runtime, and a total of O(log *n*) recursions are required, thus the runtime of this algorithm is O(*n* * log *n*). A merge sort can also be modified for performance on lists that are nearly sorted to begin with. After sorting each half of the data, if the highest element in one list is less than the lowest element in the other half, then the merge step is unnecessary. (The Java API implements this particular optimization, for instance.) The data, as the process is called recursively, might look like this:

```
{18, 6, 9, 1, 4, 15, 12, 5, 6, 7, 11}
{18, 6, 9, 1, 4} {15, 12, 5, 6, 7, 11}
{18, 6} {9, 1, 4} {15, 12, 5} {6, 7, 11}
{18} {6} {9} {1, 4} {15} {12, 5} {6} {7, 11}
{18} {6} {9} {1} {4} {15} {12} {5} {6} {7} {11}
{18} {6} {9} {1, 4} {15} {5, 12} {6} {7, 11}
{6, 18} {1, 4, 9} {5, 12, 15} {6, 7, 11}
{1, 4, 6, 9, 18} {5, 6, 7, 11, 12, 15}
{1, 4, 5, 6, 6, 7, 9, 11, 12, 15, 18}
```

Apart from being fairly efficient, a merge sort has the advantage that it can be used to solve other problems, such as determining how “unsorted” a given list is.

Heap Sort

In a heap sort, we create a heap data structure. A heap is a data structure that stores data in a tree such that the smallest (or largest) element is always the root node. (Heaps, also known as priority queues, were discussed in more detail in Data Structures.) To perform a heap sort, all data from a list is inserted into a heap, and then the root element is repeatedly removed and stored back into the list. Since the root element is always the smallest element, the result is a sorted list. If you already have a Heap implementation available or you utilize the Java PriorityQueue (newly available in version 1.5), performing a heap sort is fairly short to code:

```
Heap h = new Heap();
for (int i = 0; i < data.Length; i++)
   h.Add(data[i]);
int[] result = new int[data.Length];
for (int i = 0; i < data.Length; i++)
   data[i] = h.RemoveLowest();
```

The runtime of a heap sort has an upper bound of O(*n* * log *n*). Additionally, its requirement for storage space is only that of storing the heap; this size is linear in proportion to the size of the list. Heap sort has the disadvantage of not being stable, and is somewhat more complicated to understand beyond just the basic implementation.

Quick Sort

A quick sort, as the name implies, is intended to be an efficient sorting algorithm. The theory behind it is to sort a list in a way very similar to how a human might do it. First, divide the data into two groups of “high” values and “low” values. Then, recursively process the two halves. Finally, reassemble the now sorted list.

```
Array quickSort(Array data) {
   if (Array.Length <= 1)
      return;
   middle = Array[Array.Length / 2];
   Array left = new Array();
   Array right = new Array();
   for (int i = 0; i < Array.Length; i++)
      if (i != Array.Length / 2) {
         if (Array[i] <= middle)
            left.Add(Array[i]);
         else
            right.Add(Array[i]);
      }
   return concatenate(quickSort(left), middle, quickSort(right));
}
```

The challenge of a quick sort is to determine a reasonable “midpoint” value for dividing the data into two groups. The efficiency of the algorithm is entirely dependent upon how successfully an accurate midpoint value is selected. In a best case, the runtime is O(n * log n). In the worst case-where one of the two groups always has only a single element-the runtime drops to O(n^2). The actual sorting of the elements might work out to look something like this:

```
{18, 6, 9, 1, 4, 15, 12, 5, 6, 7, 11}
{6, 9, 1, 4, 12, 5, 6, 7, 11} {15} {18}
{6, 9, 1, 4, 5, 6, 7, 11} {12} {15} {18}
{1, 4} {5} {6, 9, 6, 7, 11} {12} {15} {18}
{1} {4} {5} {6} {6} {9, 7, 11} {12} {15} {18}
{1} {4} {5} {6} {6} {7} {9, 11} {12} {15} {18}
{1} {4} {5} {6} {6} {7} {9} {11} {12} {15} {18}
```

If it is known that the data to be sorted all fit within a given range, or fit a certain distribution model, this knowledge can be used to improve the efficiency of the algorithm by choosing midpoint values that are likely to divide the data in half as close to evenly as possible. A generic algorithm that is designed to work without respect to data types or value ranges may simply select a value from the unsorted list, or use some random method to determine a midpoint.

Radix Sort

The radix sort was designed originally to sort data without having to directly compare elements to each other. A radix sort first takes the least-significant digit (or several digits, or bits), and places the values into buckets. If we took 4 bits at a time, we would need 16 buckets. We then put the list back together, and have a resulting list that is sorted by the least significant radix. We then do the same process, this time using the second-least significant radix. We lather, rinse, and repeat, until we get to the most significant radix, at which point the final result is a properly sorted list.

For example, let’s look at a list of numbers and do a radix sort using a 1-bit radix. Notice that it takes us 4 steps to get our final result, and that on each step we setup exactly two buckets:

```
{6, 9, 1, 4, 15, 12, 5, 6, 7, 11}
{6, 4, 12, 6} {9, 1, 15, 5, 7, 11}
{4, 12, 9, 1, 5} {6, 6, 15, 7, 11}
{9, 1, 11} {4, 12, 5, 6, 6, 15, 7}
{1, 4, 5, 6, 6, 7} {9, 11, 12, 15}
```

Let’s do the same thing, but now using a 2-bit radix. Notice that it will only take us two steps to get our result, but each step requires setting up 4 buckets:

```
{6, 9, 1, 4, 15, 12, 5, 6, 7, 11}
{4, 12} {9, 1, 5} {6, 6} {15, 7, 11}
```

```
{1} {4, 5, 6, 6, 7} {9, 11} {12, 15}
```

Given the relatively small scope of our example, we could use a 4-bit radix and sort our list in a single step with 16 buckets:

```
{6, 9, 1, 4, 15, 12, 5, 6, 7, 11}
{1} {} {} {4} {5} {6, 6} {7} {} {9} {} {11} {12} {} {} {15}
```

Notice, however, in the last example, that we have several empty buckets. This illustrates the point that, on a much larger scale, there is an obvious ceiling to how much we can increase the size of our radix before we start to push the limits of available memory. The processing time to reassemble a large number of buckets back into a single list would also become an important consideration at some point.

Because radix sort is qualitatively different than comparison sorting, it is able to perform at greater efficiency in many cases. The runtime is O(*n* * *k*), where *k* is the size of the key. (32-bit integers, taken 4 bits at a time, would have *k* = 8.) The primary disadvantage is that some types of data may use very long keys (strings, for instance), or may not easily lend itself to a representation that can be processed from least significant to most-significant. (Negative floating-point values are the most commonly cited example.)

Sorting Libraries

Nowadays, most programming platforms include runtime libraries that provide a number of useful and reusable functions for us. The .NET framework, Java API, and C++ STL all provide some built-in sorting capabilities. Even better, the basic premise behind how they work is similar from one language to the next.

For standard data types such as scalars, floats, and strings, everything needed to sort an array is already present in the standard libraries. But what if we have custom data types that require more complicated comparison logic? Fortunately, object-oriented programming provides the ability for the standard libraries to solve this as well.

In both Java and C# (and VB for that matter), there is an interface called Comparable (IComparable in .NET). By implementing the IComparable interface on a user-defined class, you add a method `int CompareTo (object other)`, which returns a negative value if less than, 0 if equal to, or a positive value if greater than the parameter. The library sort functions will then work on arrays of your new data type.

Additionally, there is another interface called Comparator (IComparer in .NET), which defines a single method `int Compare (object obj1, object obj2)`, which returns a value indicating the results of comparing the two parameters.

The greatest joy of using the sorting functions provided by the libraries is that it saves a lot of coding time, and requires a lot less thought and effort. However, even with the heavy lifting already completed, it is still nice to know how things work under the hood.

More Resources

Member Tutorials

Read more than 40 data science tutorials written by topcoder members.

Problem Set Analysis

Read editorials explaining the problem and solution for each Single Round Match (SRM).

Data Science Guide

New to topcoder's data science track? Read this guide for an overview on how to get started in the arena and how competitions work.

Help Center

Need specifics about the process or the rules? Everything you need to know about competing at topcoder can be found in the Help Center.

Member Forums

Join your peers in our member forums and ask questions from the real experts - topcoder members!