Special Topic 14.3

The Quicksort Algorithm

Quicksort is a commonly used algorithm that has the advantage over merge sort that no temporary arrays are required to sort and merge the partial results.

The quicksort algorithm, like merge sort, is based on the strategy of divide and conquer. To sort a range `a[from] . . . a[to]` of the array `a`, first rearrange the elements in the range so that no element in the range `a[from] . . . a[p]` is larger than any element in the range `a[p + 1] . . . a[to]`. This step is called *partitioning* the range.

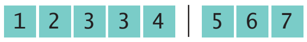
For example, suppose we start with a range

5	3	2	6	4	1	3	7

Here is a partitioning of the range. Note that the partitions aren't yet sorted.

3	3	2	1	4	|	6	5	7

You'll see later how to obtain such a partition. In the next step, sort each partition, by recursively applying the same algorithm on the two partitions. That sorts the entire range, because the largest element in the first partition is at most as large as the smallest element in the second partition.

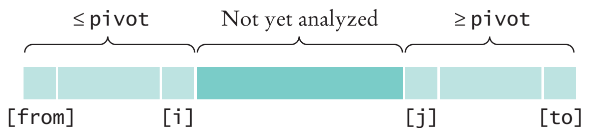


Quicksort is implemented recursively as follows:

```
public void sort(int from, int to)
{
   if (from >= to) return;
   int p = partition(from, to);
   sort(from, p);
   sort(p + 1, to);
}
```

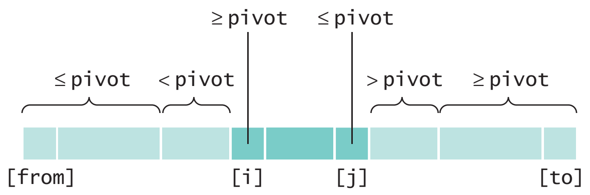
Let us return to the problem of partitioning a range. Pick an element from the range and call it the *pivot*. There are several variations of the quicksort algorithm. In the simplest one, we'll pick the first element of the range, a[from], as the pivot.

Now form two regions a[from] . . . a[i], consisting of values at most as large as the pivot and a[j] . . . a[to], consisting of values at least as large as the pivot. The region a[i + 1] . . . a[j - 1] consists of values that haven't been analyzed yet. (See the figure below.) At the beginning, both the left and right areas are empty; that is, i = from - 1 and j = to + 1.



Partitioning a Range

Then keep incrementing i while a[i] < pivot and keep decrementing j while a[j] > pivot. The figure below shows i and j when that process stops.



Extending the Partitions

Now swap the values in positions i and j, increasing both areas once more. Keep going while i < j. Here is the code for the partition method:

```
private int partition(int from, int to)
{
   int pivot = a[from];
   int i = from - 1;
   int j = to + 1;
   while (i < j)
   {
      i++; while (a[i] < pivot) i++;
      j--; while (a[j] > pivot) j--;
      if (i < j) swap(i, j);
   }
   return j;
}
```

On average, the quicksort algorithm is an $O(n \log(n))$ algorithm. Because it is simpler, it runs faster than merge sort in most cases. There is just one unfortunate aspect to the quicksort algorithm. Its *worst-case* run-time behavior is $O(n^2)$. Moreover, if the pivot element is chosen as the first element of the region, that worst-case behavior occurs when the input set is already sorted—a common situation in practice. By selecting the pivot element more cleverly, we can make it extremely unlikely for the worst-case behavior to occur. Such "tuned" quicksort algorithms are commonly used, because their performance is generally excellent. For example, the `sort` method in the `Arrays` class uses a quicksort algorithm.

Another improvement that is commonly made in practice is to switch to insertion sort when the array is short, because the total number of operations of insertion sort is lower for short arrays. The Java library makes that switch if the array length is less than 7.