Tutorial 3: *Working with Arrays, Loops, and Conditional Statements*

Sorting Algorithms

In computer science and mathematics, a sorting algorithm is an algorithm that puts elements of a list in a certain order. Here are some **s**ummaries of popular sorting algorithms**.**

Bubble sort

Bubble sort is a straightforward and simplistic method of sorting data that is used in computer science education. The algorithm starts at the beginning of the data set. It compares the first two elements, and if the first is greater than the second, it swaps them. It continues doing this for each pair of adjacent elements to the end of the data set. It then starts again with the first two elements, repeating until no swaps have occurred on the last pass. Although simple, this algorithm is highly inefficient and is rarely used except in education. A slightly better variant, cocktail sort, works by inverting the ordering criteria and the pass direction on alternating passes.

Selection sort

Selection sort is a simple sorting algorithm that improves on the performance of bubble sort. It works by first finding the smallest element using a linear scan and swapping it into the first position in the list, then finding the second smallest element by scanning the remaining elements, and so on. Selection sort is unique compared to almost any other algorithm in that its running time is not affected by the prior ordering of the list: it performs the same number of operations because of its simple structure. Selection sort also requires only *n* swaps, and hence just Î˜(*n*) memory writes, which is optimal for any sorting algorithm. Thus it can be very attractive if writes are the most expensive operation, but otherwise selection sort will usually be outperformed by insertion sort or the more complicated algorithms.

Insertion sort

Insertion sort is a simple sorting algorithm that is relatively efficient for small lists and mostly-sorted lists, and often is used as part of more sophisticated algorithms. It works by taking elements from the list one by one and inserting them in their correct position into a new sorted list. In arrays, the new list and the remaining elements can share the array's space, but insertion is expensive, requiring shifting all following elements over by one. The insertion sort works just like its name suggests - it inserts each item into its proper place in the final list. The simplest implementation of this requires two list structures - the source list and the list into which sorted items are inserted. To save memory, most implementations use an in-place sort that works by moving the current item past the already sorted items and repeatedly swapping it with the preceding item until it is in place. Shell sort (see below) is a variant of insertion sort that is more efficient for larger lists. This method is much more efficient than the bubble sort, though it has more constraints.

Shell sort

Shell sort was invented by Donald Shell in 1959. It improves upon bubble sort and insertion sort by moving out of order elements more than one position at a time. One implementation can be described as arranging the data sequence in a two-dimensional array and then sorting the columns of the array using insertion sort. Although this method is inefficient for large data sets, it is one of the fastest algorithms for sorting small numbers of elements (sets with less than 1000 or so elements). Another advantage of this algorithm is that it requires relatively small amounts of memory.

Merge sort

Merge sort takes advantage of the ease of merging already sorted lists into a new sorted list. It starts by comparing every two elements (*i.e.* 1 with 2, then 3 with 4...) and swapping them if the first should come after the second. It then merges each of the resulting lists of two into lists of four, then merges those lists of four, and so on; until at last two lists are merged into the final sorted list. Of the algorithms described here, this is the first that scales well to very large lists.

Heapsort

Heapsort is a much more efficient version of selection sort. It also works by determining the largest (or smallest) element of the list, placing that at the end (or beginning) of the list, then continuing with the rest of the list, but accomplishes this task efficiently by using a data structure called a heap, a special type of binary tree. Once the data list has been made into a heap, the root node is guaranteed to be the largest element. It is removed and placed at the end of the list, then the heap is rearranged so the largest element remaining moves to the root . Using the heap, finding the next largest element takes *O(log n)* time, instead of *O(n)* for a linear scan as in simple selection sort. This allows Heapsort to run in *O(n log n)* time.

Quicksort

Quicksort is a divide and conquer algorithm which relies on a *partition* operation: to partition an array, we choose an element, called a *pivot*, move all smaller elements before the pivot, and move all greater elements after it. This can be done efficiently in linear time and in-place. We then recursively sort the lesser and greater sublists. Efficient implementations of quicksort (with in-place partitioning) are typically unstable sorts and somewhat complex, but are among the fastest sorting algorithms in practice. Together with its modest O(log n) space usage, this makes quicksort one of the most popular sorting algorithms, available in many standard libraries. The most complex issue in quicksort is choosing a good pivot element; consistently poor choices of pivots can result in drastically slower (O(n^2)) performance, but if at each step we choose the *median* as the pivot then it works in O(n log n).

Bucket sort

Bucket sort is a sorting algorithm that works by partitioning an array into a finite number of buckets. Each bucket is then sorted individually, either using a different sorting algorithm, or by recursively applying the bucket sorting algorithm.

Radix sort

Radix sort is an algorithm that sorts a list of fixed-size numbers of length k in O(n Â· k) time by treating them as bit strings. We first sort the list by the least significant bit while preserving their relative order using a stable sort. Then we sort them by the next bit, and so on from right to left, and the list will end up sorted. Most often, the counting sort algorithm is used to accomplish the bitwise sorting, since the number of values a bit can have is small.