Sorting

What is sorting in Computer Science?

Simply put, arranging data in an ordered sequence is called "sorting". In Computer Science, the more sophisticated and universal concept of sorting has to do with the technique (e.g., algorithm) for placing arrays in increasing or decreasing order.

In computer science, 'sorting' usually refers to bringing a set of items into some well-defined order. To be able to do this, we first need to specify the notion of *order* on the items we are considering. For example, for numbers we can use the usual numerical order (that is, defined by the mathematical 'less than' or '<' relation) and for strings the so-called *lexicographic* or *alphabetic* order, which is the one dictionaries and encyclopedias use.

Usually, what is meant by *sorting* is that once the sorting process is finished, there is a simple way of 'visiting' all the items in order, for example to print out the contents of a database. This may well mean different things depending on how the data is being stored.

For example, if all the objects are sorted and stored in an array a of size n, then:

```
for i = 0…,n-1
   print(a[i])
```

would print the items in ascending order.

The Problem of Sorting

Sorting is important because having the items in order makes it much easier to *find* a given item, such as the cheapest item or the file corresponding to a particular student. It is thus closely related to the problem of *search*, as we will see next week.

In your computer studies, you have already seen that, by having the data items stored in a sorted array, the average (and worst case) complexity of searching for a particular item can be

reduced. So, if you often need to look up items, it is worth the effort to sort the whole collection first. Imagine using a dictionary or phone book in which the entries do not appear in some known logical order.

With this line of reasoning, it then follows that sorting algorithms are important tools for program designers. Different algorithms are suited to different situations, and you shall see that there is no 'best' sorting algorithm for everything, and therefore a few of them will be introduced in our lesson this week.

Common Sorting Strategies

One way of organizing the various sorting algorithms is by classifying the underlying idea, or 'strategy'. Some of the key strategies are:

enumeration sorting	Consider all items. If you know that there are N items which are smaller than the one you are currently considering, then its final position will be at number $N + 1$.
exchange sorting	If two items are found to be out of order, exchange them. Repeat till all items are in order.
selection sorting	Find the smallest item, put it in the first position, find the smallest of the remaining items, put it in the second position . . .
insertion sorting	Take the items one at a time and insert them into an initially empty data structure such that the data structure continues to be sorted at each stage.
divide and conquer	Recursively split the problem into smaller sub-problems till you just have single items that are trivial to sort. Then put the sorted 'parts' back together in a way that preserves the sorting.

All these strategies are based on *comparing* items and then rearranging them accordingly. These are known as comparison-based sorting algorithms.

The sorting strategies noted above are based on the assumption that all the items to be sorted will fit into the computer's internal memory, which is why they are often referred to as being *internal sorting algorithms*. If the whole set of items cannot be stored in the internal memory at one time, different techniques need be used. These days, given the growing power and memory of computers, external storage is becoming much less commonly needed when sorting, so we will not consider *external sorting algorithms*. A future course on filesystems will provide an opportunity for you to explore external sort techniques in more detail.

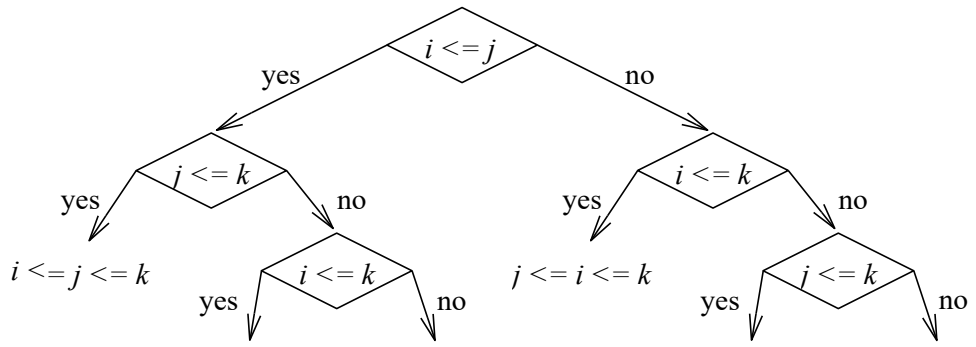
How Many Comparisons will it Take?

A straightforward way to compute the *time complexity* of sorting algorithms is to count the number of comparisons they need to carry out, as a function of the number of items to be

sorted. W are generally most interested in having a *lower bound* for the number of comparisons needed for the best algorithm in the worst case. It is not that the upper bound consideration is a moot point in the picture. Rather, it is the lower bound (i.e. faster algorithm) that we aspire to achieve. In other words, we want to know the minimum number of comparisons required to have all the information needed to sort an arbitrary collection of items. Then we can see how well particular sorting algorithms compare against that theoretical lower bound.

For sorting algorithms based on comparisons, however, it turns out that a tight lower bound does exist. Consider, even if the given collection of items is already sorted, we must still check all the items one at a time to see whether they are in the correct order. Thus, the lower bound must be at least n, the number of items to be sorted, since we need at least n steps to examine every element. If we already knew a sorting algorithm that works in n steps, then we could stop looking for a better algorithm: n would be both a lower bound and an upper bound to the minimum number of steps, and hence an *exact bound* . However, as we shall shortly see, no algorithm can actually take fewer than $O(n\log_2 n)$ comparisons in the worst case. If, in addition, we can design an algorithm that works in $O(n\log_2 n)$ steps, then we will have obtained an exact bound. We shall start by demonstrating that every algorithm needs at least $O(n\log_2 n)$ comparisons.

To begin with, let us assume that we only have three items, i, j, and k. If we have found that $i \leq j$ and $j \leq k$, then we know that the sorted order is: i, j, k. So it took us two comparisons to find this out. In some cases, however, it is clear that we will need as many as three comparisons. For example, if the first two comparisons tell us that $i > j$ and $j \leq k$, then we know that j is the smallest of the three items, but we cannot say from this information how i and k relate. A third comparison is needed. So what is the *average* and *worst* number of comparisons that are needed? This can best be determined from the so-called *decision tree*, where we keep track of the information gathered so far and count the number of comparisons needed. The decision tree for the three item example we were discussing is:



So what can we deduce from this about the general case? The decision tree will obviously always be a binary tree. It is also clear that its *height* will tell us how many comparisons will be needed in the worst case, and that the average length of a path from the root to a leaf will

give us the average number of comparisons required. The leaves of the decision tree are height h is at most 2^h, so we want to find h such that

$$2^h \geq n! \quad \text{or} \quad h \geq \log_2(n!)$$

There are numerous approximate expressions that have been derived for $\log_2(n!)$ for large n, but they all have the same dominant term, namely $n\log_2 n$. (We will learn that, when talking about time complexity, we ignore any sub-dominant terms and constant factors.) Therefore, no sorting algorithm based on comparing items can have a better average or worst case performance than using a number of comparisons that is approximately $n\log_2 n$ for large n. It remains to be seen whether this $O(n\log_2 n)$ complexity can actually be achieved in practice. To do this, we would have to exhibit at least one algorithm with this performance behavior (and convince ourselves that it really does have this behavior). In fact, you will see in your Computer Science curriculum that there are several algorithms with this behavior.

We shall proceed now by looking in turn at a few specific sorting algorithms of increasing sophistication, that involve the various strategies listed above. The way they work depends on what kind of data structure contains the items we wish to sort. We start with approaches that work with simple arrays, and then move on to using more complex data structures that lead to more efficient algorithms.

This week we will look at **selection sort** and a divide-and-conquer algorithm known as **merge sort.**

Selection Sort

Selection Sort is a form of *selection sorting*. It first finds the smallest item and puts it into a[0] by exchanging it with whichever item is in that position already. Then it finds the second-smallest item and exchanges it with the item in a[1]. It continues this way until the whole array is sorted. More generally, at the ith stage, Selection Sort finds the ith-smallest item and swaps it with the item in a[i-1]. There is no need to check for the ith-smallest item in the first $i-1$ elements of the array.

For the example starting array | 4 | 1 | 3 | 2 |, Selection Sort first finds the smallest item in the whole array, which is a[1]=1, and swaps this value with that in a[0] giving | 1 | 4 | 3 | 2 |. Then, for the second step, it finds the smallest item in the reduced array a[1],a[2],a[3], that is a[3]=2, and swaps that into a[1], giving | 1 | 2 | 3 | 4 |. Finally, it finds the smallest of the reduced array a[2],a[3], that is a[2]=3, and swaps that into a[2], or recognizes that a swap is not needed, giving | 1 | 2 | 3 | 4 |.

The general algorithm for Selection Sort can be written:

```
for ( i = 0 ; i < n-1 ; i++ ) {
    k = i
    for ( j = i+1 ; j < n ; j++ )
        if ( a[j] < a[k] )
            k = j
    swap a[i] and a[k]
}
```

The outer loop goes over the first n − 1 positions to be filled, and the inner loop goes through the currently unsorted portion to find the next smallest item to fill the next position. Note that there is exactly one swap for each iteration of the outer loop.

The time complexity is again the number of comparisons carried out. The outer loop is carried out $n-1$ times. In the inner loop, which is carried out $(n-1)-i = n-1-i$ times, one comparison occurs.

Hence the total number of comparisons is:

$$\sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} (n-1-i)$$
$$= (n-1) + \cdots + 2 + 1$$
$$= \frac{n(n-1)}{2}.$$

Therefore, the number of comparisons for Selection Sort is proportional to n^2, in the worst case as well as in the average case, and hence the average and worst-case time complexities are both $O(n^2)$.

Note that Selection Sort involves two nested for loops over $O(n)$ items, so it is easy to see that the overall complexities will be $O(n^2)$ without having to compute the exact number of comparisons.

Sorting Algorithm Stability

One often wants to sort items which might have identical keys (e.g., ages in years) in such a way that items with identical keys are kept in their original order, particularly if the items have already been sorted according to different criteria (e.g., alphabetical). So, if we denote the original order of an array of items by subscripts, we want the subscripts to end up in order for each set of items with identical keys. For example, if we start out with the array $[5_1, 4_2, 6_3, 5_4, 6_5, 7_6, 5_7, 2_8, 9_9]$, it should be sorted to $[2_8, 4_2, 5_1, 5_4, 5_7, 6_3, 6_5, 7_6, 9_9]$ and not to $[2_8, 4_2, 5_4, 5_1, 5_7, 6_3, 6_5, 7_6, 9_9]$. Sorting algorithms which satisfy this useful property of ensuring parallel arrays are said to be *stable*.

The easiest way to determine whether a given algorithm is stable is to consider whether the algorithm can ever swap identical items past each other. In this way, the stability of the sorting algorithms studied so far can easily be established:

Selection Sort This is not stable, because there is nothing to stop an item being swapped past another item that has an identical key. For example, the array $[2_1, 2_2, 1_3]$ would be sorted to $[1_3, 2_2, 2_1]$ which has items 2_2 and 2_1 in the wrong order.

The issue of sorting stability needs to be considered when developing more complex sorting algorithms. Often there are stable and non-stable versions of the algorithms, and one has to consider whether the extra cost of maintaining stability is worth the effort.

Divide and Conquer Algorithms

The sorting algorithm considered so far work on the whole set of items together. Instead, *divide and conquer* algorithms **recursively** split the sorting problem into more manageable sub-problems. The idea is that it will usually be easier to sort many smaller collections of items than one big one, and sorting single items is trivial. So we repeatedly split the given collection into two smaller parts until we reach the 'base case' of one-item collections, which require no effort to sort, and then merge them back together again. There are two main approaches for doing this:

Assuming we are working on an array a of size n with entries `a[0]`,…,`a[n-1]`, then the obvious approach is to simply split the set of indices. That is, we split the array at item $n/2$

and consider the two sub-arrays a[0],...,a[(n-1)/2] and a[(n+1)/2],...,a[n-1]. This method has the advantage that the splitting of the collection into two collections of equal (or nearly equal) size at each stage is easy. However, the two sorted arrays that result from each split have to be *merged* together carefully to maintain the ordering. This is the underlying idea for a sorting algorithm called *mergesort*.

Merge Sort

The divide and conquer sorting strategy based on repeatedly splitting the array of items into two sub-arrays is called *mergesort*. This simply splits the array at each stage into its first and last half, without any reordering of the items in it. However, that will obviously not result in a set of sorted sub-arrays that we can just append to each other at the end. So mergesort needs another procedure merge that merges two sorted sub-arrays into another sorted array. Integer variables left and right can be used to refer to the lower and upper index of the relevant array, and mid refers to the end of its left sub-array. Thus a suitable mergesort algorithm is:

```
mergesort(array a, int left, int right) { if
   ( left < right ) {
      mid = (left + right) / 2
      mergesort(a, left, mid)
      mergesort(a, mid+1,right)
      merge(a, left, mid, right)
   }
}
```

The Merge Algorithm

The principle of merging two sorted collections (whether they be lists, arrays, or something else) is quite simple: Since they are sorted, it is clear that the smallest item overall must be either the smallest item in the first collection or the smallest item in the second collection. Let us assume it is the smallest key in the first collection. Now the second smallest item overall must be either the second-smallest item in the first collection, or the smallest item in the second collection, and so on. In other words, we just work through both collections and at each stage, the 'next' item is the current item in either the first or the second collection.

The implementation will be quite different, however, depending on which data structure we are using. When arrays are used, it is actually necessary for the merge algorithm to create a new array to hold the result of the operation at least temporarily. In contrast, when using linked lists, it would be possible for merge to work by just changing the reference to the next node. This does make for somewhat more confusing code, however.

For arrays, a suitable *merge* algorithm would start by creating a new array b to store the results, then repeatedly add the next smallest item into it until one sub-array is finished, then copy the remainder of the unfinished sub-array, and finally copy b back into a:

```
merge(array a, int left, int mid, int right) {
    create new array b of size right-left+1
    bcount = 0
    lcount = left
    rcount = mid+1
    while ( (lcount <= mid) and (rcount <= right) ) { if
        ( a[lcount] <= a[rcount] )
            b[bcount++] = a[lcount++] else
            b[bcount++] = a[rcount++]
    }
    if ( lcount > mid )
        while ( rcount <= right )
            b[bcount++] = a[rcount++]
    else
        while ( lcount <= mid )
            b[bcount++] = a[lcount++]
    for ( bcount = 0 ; bcount < right-left+1 ; bcount++ )
        a[left+bcount] = b[bcount]
}
```

The merge algorithm never swaps identical items past each other, and the splitting does not change the ordering at all, so the whole Merge Sort algorithm is *stable*.

Complexity of Merge Sort. The total number of comparisons needed at each recursion level of Merge Sort is the number of items needing merging which is $O(n)$, and the number of recursions needed to get to the single item level is $O(\log_2 n)$, so the total number of comparisons and its time complexity are $O(n \log_2 n)$. This holds for the worst case as well as the average case. It is possible to speed up Merge Sort by abandoning the recursive algorithm when the sizes of the sub-collections become small. For arrays, 16 would once again be a suitable size to switch to an algorithm like Selection Sort.