

Lecture Notes 13 – Sorting

CSS 501 - Data Structures and Object-Oriented Programming – Professor Clark F. Olson

Reading for this lecture: Carrano, Chapter 11

Introduction

Today we will be discussing various sorting algorithms and examining their efficiency. Sorting is one of the most important operations in computer science, since it is very common to want a collection of data to be in some order. For example, we saw previously that binary search could find an item in a list much faster than sequential search, but only if the list is sorted. When the collection of data that we are sorting is large, it is very important that an efficient algorithm is used, since we would otherwise spend more time than necessary sorting. The algorithms we will discuss can be applied to any type of objects, including integers, floating point numbers, strings, and complex objects.

As long as there is some way to order the objects (the *sort key*), any type can be sorted. We will assume that the comparison operators ($<$, $>$, $==$) have been previously defined for whatever type we are working with. We will also assume that we have a function to swap elements of our data type:

```
template <typename Object>
void swap(Object &a, Object &b) {

    Comparable tmp = a;
    a = b;
    b = tmp;
}
```

Selection sort

Selection sort works by finding the item that goes at the end of the list (or the beginning), moving that item to the correct location, and then sorting the rest of the items (either iteratively or recursively). For example, if we start with: 3 2 4 5 1. The first step is to find the largest (5), and swap it with the last element yielding: 3 2 4 1 5. The next step finds the next largest and swaps it with the next to last position: 3 2 1 4 5. This continues until the entire array is sorted. Here is a recursive version:

```
template <typename Comparable>
void selectionSort(vector<Comparable> &a, int last) {
    if (last <= 0) return;

    Comparable largest = a[0];
    int index = 0;

    for (int i = 1; i <= last; i++)
        if (a[i] > largest) {
            largest = a[i];
            index = i;
        }

    swap(a[index], a[last]);
    selectionSort(a, last - 1);
}
```

If we start with an array containing 3 2 4 5 1, the first function call will move the 5 to the last position, the second will move the 4 to the next-to-last position, and so on.

How long does this take? Sorting algorithms are often evaluated using **the number of comparisons** that are performed between elements. For **each comparison**, we never do more than **$O(1)$ work**. The for-loop is executed $n - 1$ times (each execution does one comparison and sometimes two assignments – $O(1)$ total time). Then, we recursively solve a problem we size $n - 1$, so our recurrence relation for the number of comparisons performed is:

$$T(n) = n - 1 + T(n - 1)$$

$$T(1) = 0$$

Using iteration, we get:

$$T(n) = (n - 1) + (n - 2) + \dots + 1 = n * (n - 1) / 2 = O(n^2)$$

This algorithm performs the same number of comparisons in the best, worst, and average case.

Bubble Sort

In bubble sort (the first sorting algorithm many people learn), **comparisons are only performed between adjacent elements** in the array. The elements are swapped if they are in the wrong order. Each pass through the data considers each pair of adjacent elements that are unsorted. After the first pass, the largest element must be in the last element of the array, so the next pass only has to consider one less element. Of course, we usually have to perform multiple passes through the data to sort the data completely.

Example: 3 2 4 5 1

We first compare the first two elements (swapping them, since they are out of order):

2 3 4 5 1

We then compare the second and third elements (not swapped), the third and fourth (not swapped) and the fourth and fifth elements (swapped)

2 3 4 1 5

This has “bubbled” the 5 “up” to the last position. Since, the array isn’t sorted yet, we have to do another pass, but this time we don’t have to go as far, since we know the largest element is in the last position. This continues until the array is sorted.

Here is an iterative version of the bubble sort algorithm:

```
template <typename Comparable>
void bubbleSort(vector<Comparable> &a) {
    bool sorted = false;

    // Pass through the data no more than a.size() - 1 times
    for (int pass = 1; pass < a.size() && !sorted; pass++) {

        // Each pass bubbles up the largest item to the top
        sorted = true;
        for (int index = 0; index < a.size() - pass; index++) {
            if (a[index] > a[index+1]) {
                swap(a[index], a[index+1]);
                sorted = false;
            }
        }
    }
}
```

Let’s look at the running time. **In the best case, the data is already sorted.** In this case, only one pass through the data is required and this pass performs $n - 1$ comparisons. This is $O(n)$ time. **In the worst case, the data is in the completely wrong order.** This requires $n - 1$ passes, since the last item can only move one spot during each pass. The first pass requires $n - 1$ comparisons, the second $n - 2$ comparisons and so on, so the worst case is the same as the selection sort: $n * (n - 1) / 2 = O(n^2)$ comparisons.

We won’t examine the **average-case** in detail, but this is also $O(n^2)$, if each order of the array elements is equally likely. Note however, that even though they have the same worst-case complexity, **bubble sort performs many more swaps than selection sort does.** Selection sort performs (at most) $n - 1$ swaps between data elements, while the bubble sort swaps $n * (n - 1) / 2$ elements in the worst case (when the list is sorted in reverse order). If the data elements are very large, this is a significant drawback.

Insertion sort

The next algorithm is insertion sort. In this method, **the first block of elements is maintained in sorted order** and the **next unsorted element is inserted into the correct spot** at each step. Let's say the list is **3 2 5 4 1**. At the beginning, we consider only the first element. This trivial list is sorted, since any list of one element is sorted. Next, we insert the second element at the correct spot. This requires swapping 3 and 2, to get **2 3 5 4 1**. Now, we insert the 5 into the sorted elements, but it is already in the correct spot. Next, the 4 gets swapped with the 5, leaving us with **2 3 4 5 1**. Finally, the 1 gets inserted, which requires shifting all of the other elements one spot to the right.

Here is code for the algorithm:

```
template <typename Comparable>
void insertionSort(vector<Comparable> &a)
{
    // Sort the next item. After each iteration, items 1..unsorted are sorted.
    for (int unsorted = 1; unsorted < a.size(); unsorted++) {
        int loc;
        Comparable nextItem = a[unsorted];

        for (loc = unsorted; loc > 0 && nextItem < a[loc - 1]; loc--)
            a[loc] = a[loc - 1];

        a[loc] = nextItem;
    }
}
```

The outer for-loop (using **unsorted**) executes $n - 1$ times. The inner for loop (using **loc**) executes at least 1 time and at most **unsorted** times. So, the total is:

Best case: $T(n) = 1 + 1 + \dots + 1 = n - 1 = O(n)$

Worst case: $T(n) = 1 + 2 + \dots + (n - 1) = n * (n - 1) / 2 = O(n^2)$

We can determine the average number of comparisons using the concept of an **inversion**. In any array, the number of inversions is any pair of numbers that are not in the correct order (no matter how far apart they are in the array). For our example case: **3 2 4 5 1**, there are **5 inversions: (3, 2), (3, 1), (2, 1), (4, 1), (5, 1)**. Note that each time we move an element in the insertion sort, this fixes exactly one inversion, and we do $O(1)$ work per move. This implies that the work we do is proportional to the number of inversions in the list. There are $n(n-1) / 2$ pairs of elements in the array. On average, half of them will be inverted if we assume that each ordering is equally likely. So, the **average number of inversions is $n(n-1) / 4$** and, on average, we do $O(n^2)$ work.

This algorithm also **moves the data a lot in the worst case**, although it does this a little more efficiently than bubble sort does, since it can wait until it gets to the right location before it writes the next unsorted value into the array.

Insertion sort is useful for some cases. One is when you must process the data incrementally. That is, one item is added at a time. For example, in a database application, one record is added at a time. If the records are stored in sorted order, then insertion sort is a good choice. Another case where insertion sort is useful is when the numbers are nearly sorted when you start (bubble sort also works well for this case).

Lower bound on simple sorts

We can prove that if we use an algorithm that only compares and swaps neighbors, the algorithm must be $O(n^2)$ on average. The reason is that we can never fix more than one inversion per swap and we have seen that there are on average $O(n^2)$ of them. We can get around this using methods that compare and swap elements that are not neighbors.

Shell sort

Shell sort is an easy to write algorithm that can do better than $O(n^2)$, but not as good as the best sorting algorithms. It is of interest primarily because it is simple to write and because the analysis of the algorithm is complex (and not completely understood). The basic idea is that, rather than comparing neighbors, we compare (and swap, if necessary) elements that are farther away (using some **gap** between the elements). This allows us to fix multiple

inversions at once, but it doesn't completely sort the elements. We can reduce the gap and continue. Eventually, the gap has to be reduced to one, at which point we are performing insertion sort, but the array should be nearly sorted by then. Here is the complete algorithm.

```
template <typename Comparable>
void shellSort(vector<Comparable> &a) {

    int gap = choose_initial_gap();
    while (gap > 0) {
        for (int unsorted = gap; unsorted < a.size(); unsorted++) {
            int loc;
            Comparable nextItem = a[unsorted];

            for (loc = unsorted; loc >= gap && nextItem < a[loc - gap]; loc-=gap)
                a[loc] = a[loc - gap];

            a[loc] = nextItem;
        }

        gap = reduce(gap);           // Reduce to what?
    }
}
```

Oddly, adding an extra loop to insertion sort – while (gap > 0) – reduces the amount of work done by the algorithm (assuming the gap is reduced appropriately). How should we reduce the gap? Here are some options:

- | | |
|--|--|
| 1. Reduce it by one at each step. | This is bad. $O(n^2)$ best and worst case. |
| 2. Divide it by two at each step. | This is ok. $O(n^2)$ worst case. $O(n^{1.5})$ average case. |
| 3. Divide it by two and add one if it is even. | This is good. $O(n^{1.5})$ worst case. $O(n^{1.25})$ average case? |
| 4. Divide it by 2.2. | This is even better. $O(n^{1.16})$ average case? |
| 5. $1, 3, 7, \dots, 2^k - 1$ | This is also $O(n^{1.5})$ in the worst case. |
| 6. $1, 3, 9, \dots, 2^k + 1$ | This is also $O(n^{1.5})$ in the worst case. |
| 7. $1, \dots, 4^k + 3 \cdot 2^{k-1} + 1$ | This is $O(n^{(4/3)})$ in the worst case. |
| 8. $1, 2, 3, 4, 6, 8, 9, 12, 16, \dots, 2^p 3^q$ | This is $O(n \log^2 n)$ in the worst case (a better complexity). |
| 9. $1, 4, 10, 23, 57, 132, 301, 701, 1750$ | This appears optimal (based on experiments) for large arrays. |

A java demo of shell sort can be found at:

<http://www.cs.princeton.edu/~rs/shell/animate.html>

Lower bound on sorting algorithms that use comparisons

We also discussed a lower bound on the complexity of sorting algorithms that use comparisons between the items being sorted. I argued informally that no such algorithms can achieve a time complexity better than $O(n \log n)$. This is discussed in further detail here:

https://en.wikipedia.org/wiki/Comparison_sort

Practice problems (optional):

Carrano, Chapter 11:

#2, #3, #4, #6, #8, #9