13 Basic Sorting Algorithms

13.1 Introduction

Sorting is one of the most fundamental and important data processing tasks.

> **Sorting algorithm**: An algorithm that rearranges records in lists so that they follow some well-defined ordering relation on values of keys in each record.

An *internal* sorting algorithm works on lists in main memory, while an *external* sorting algorithm works on lists stored in files. Some sorting algorithms work much better as internal sorts than external sorts, but some work well in both contexts. A sorting algorithm is *stable* if it preserves the original order of records with equal keys.

Many sorting algorithms have been invented; in this chapter we will consider the simplest sorting algorithms. In our discussion in this chapter, all measures of input size are the length of the sorted lists (arrays in the sample code), and the basic operation counted is comparison of list elements (also called *keys*).

13.2 Bubble Sort

One of the oldest sorting algorithms is bubble sort. The idea behind it is to make repeated passes through the list from beginning to end, comparing adjacent elements and swapping any that are out of order. After the first pass, the largest element will have been moved to the end of the list; after the second pass, the second largest will have been moved to the penultimate position; and so forth. The idea is that large values "bubble up" to the top of the list on each pass.

A Ruby implementation of bubble sort appears in Figure 1.

```
def bubble_sort(array)
  (array.size-1).downto(1).each do | j |
    1.upto(j).each do | i |
      if array[i] < array[i-1]
        array[i], array[i-1] = array[i-1], array[i]
      end
    end
  end
  return array
end
```

Figure 1: Bubble Sort

It should be clear that the algorithm does exactly the same key comparisons no matter what the contents of the array, so we need only consider its every-case complexity.

On the first pass through the data, every element in the array but the first is compared with its predecessor, so n-1 comparisons are made. On the next pass, one less comparison is made, so n-2 comparisons are made. This continues until the last pass, where only one comparison is made. The total number of comparisons is thus given by the following summation.

$$C(n) = \sum_{i=1 \text{ to } n-1} i = n(n-1)/2$$

Clearly, $n(n-1)/2 \in O(n^2)$.

Bubble sort is not very fast. Various suggestions have been made to improve it. For example, a Boolean variable can be set to false at the beginning of each pass through the list and set to true whenever a swap is made. If the flag is false when the pass is completed, then no swaps were done and the array is sorted, so the algorithm can halt. This gives exactly the same worst case complexity, but a best case complexity of only n. The average case complexity is still in $O(n^2)$, however, so this is not much of an improvement.

13.3 Selection Sort

The idea behind selection sort is to make repeated passes through the list, each time finding the largest (or smallest) value in the unsorted portion of the list, and placing it at the end (or beginning) of the unsorted portion, thus shrinking the unsorted portion and growing the sorted portion. The algorithm works by repeatedly "selecting" the item that goes at the end of the unsorted portion of the list.

A Ruby implementation of selection sort appears in Figure 2.

```
def selection_sort(array)
  0.upto(array.size-2).each do | j |
    min_index = j
    (j+1).upto(array.size-1).each do | i |
      min_index = i if array[i] < array[min_index]
    end
    array[j], array[min_index] = array[min_index], array[j]
  end
  return array
end
```

Figure 2: Selection Sort

This algorithm finds the minimum value in the unsorted portion of the list n-1 times and puts it where it belongs. Like bubble sort, it does exactly the same thing no matter what the contents of the array, so we need only consider its every-case complexity.

On the first pass through the list, selection sort makes n-1 comparisons; on the next pass, it makes n-2 comparisons; on the third, it makes n-3 comparisons, and so forth. It makes n-1 passes altogether, so its complexity is

$$C(n) = \Sigma_{i=1 \text{ to } n-1} i = n(n-1)/2$$

As noted before, $n(n-1)/2 \in O(n^2)$.

Although the number of comparisons that selection sort makes is identical to the number that bubble sort makes, selection sort usually runs considerable faster. This is because bubble sort typically makes many swaps on every pass through the list while selection sort makes only one. Nevertheless, neither of these sorts is particularly fast.

13.4 Insertion Sort

Insertion sort works by repeatedly taking an element from the unsorted portion of a list and inserting it into the sorted portion of the list until every element has been inserted. This algorithm is the one usually used by people when sorting piles of papers.

A Ruby implementation of insertion sort appears in Figure 3.

```
def insertion_sort(array)
  1.upto(array.size-1).each do | j |
    element = array[j]
    i = j
    while 0 < i && element < array[i-1]
      array[i] = array[i-1]
      i -= 1
    end
    array[i] = element
  end
  return array
end
```

Figure 3: Insertion Sort

A list with only one element is already sorted, so the elements inserted begin with the second element in the array. The inserted element is held in the `element` variable and values in the sorted portion of the array are moved up to make room for the inserted element in the same loop that finds the right place to make the insertion. Once that spot is found, the loop ends and the inserted element is placed into the sorted portion of the array.

Insertion sort does different things depending on the contents of the list, so we must consider its worst, best, and average case behavior. If the list is already sorted, one comparison is made for each of n-1 elements as they are "inserted" into their current locations. So the best case behavior of insertion sort is

$$B(n) = n-1$$

The worst case occurs when every inserted element must be placed at the beginning of the already sorted portion of the list; this happens when the list is in reverse order. In this case, the first element inserted requires one comparison, the second two, the third three, and so forth, and n-1 elements must be inserted. Hence

$$W(n) = \sum_{i=1 \text{ to } n-1} i = n(n-1)/2$$

To compute the average case complexity, let's suppose that the inserted element is equally likely to end up at any location in the sorted portion of the list, as well as the position it initially occupies. When inserting the element with index j, there are j+1 locations where the element may be inserted, so the probability of inserting into each location is 1/(j+1). Hence the average number of comparison to insert the element with index j is given by the following expression.

$$\begin{aligned}
&1/(j+1) + 2/(j+1) + 3/(j+1) + \ldots + j/(j+1) + j/(j+1) \\
&= 1/(j+1) \cdot \Sigma_{i=1 \text{ to } j}\, i + j/(j+1) \\
&= 1/(j+1) \cdot j(j+1)/2 + j/(j+1) \\
&= j/2 + j/(j+1) \\
&\approx j/2 + 1
\end{aligned}$$

The quantity $j/(j+1)$ is always less than one and it is very close to one for large values of j, so we simplify the expression as noted above to produce a close upper bound for the count of the average number of comparisons done when inserting the element with index j. We will use this simpler expression in our further computations because we know that the result will always be a close upper bound on the number of comparisons.

We see that when inserting an element into the sorted portion of the list we have to make comparisons with about half the elements in that portion of the list, which makes sense.

Armed with this fact, we can now write down an equation for the approximate average case complexity:

$$\begin{aligned}
A(n) &= \Sigma_{j=1 \text{ to } n-1} (j/2 + 1) \\
&= ½\, \Sigma_{j=1 \text{ to } n-1}\, j + \Sigma_{j=1 \text{ to } n-1}\, 1 \\
&= ½\, (n(n-1)/2) + (n-1) \\
&= (n^2 + 3n - 4)/4
\end{aligned}$$

In the average case, insertion sort makes about half as many comparisons as it does in the worst case. Unfortunately, both these functions are in $O(n^2)$, so insertion sort is not a great sort. Nevertheless, insertion sort is quite a bit better than bubble and selection sort on average and in the best case, so it is the best of the three $O(n^2)$ sorting algorithms.

Insertion sort has one more interesting property to recommend it: it sorts nearly sorted lists very fast. A *k-nearly sorted list* is a list all of whose elements are no more than k positions from their final locations in the sorted list. Inserting any element into the already sorted portion of the list requires at most k comparisons. A close upper bound on the worst case complexity of insertion sort on a k-nearly sorted list is:

$$W(n) = \sum_{i=1 \text{ to } n-1} k = k \cdot (n-1)$$

Because k is a constant, $W(n)$ is in $O(n)$, that is, insertion sort always sorts a nearly sorted list in linear time, which is very fast indeed.

13.5 Shell Sort

Shell sort is an interesting variation of insertion sort invented by Donald Shell in 1959. It works by insertion sorting the elements in a list that are h positions apart for some h, then decreasing h and doing the same thing over again until $h = 1$.

A version of Shell sort in Ruby appears in Figure 4.

```
def shell_sort(array)
  # compute the starting value of h
  h = 1;
  h = 3*h + 1 while h < a.size/9

  # insertion sort using decreasing values of h
  while 0 < h do
    h.upto(array.size-1).each do | j |
      element = array[j]
      i = j
      while 0 < i && element < array[i-h]
        array[i] = array[i-h]
        i -= h
      end
      array[i] = element
    end
    h /= 3
  end
  return array
end
```

Figure 4: Shell Sort

Although Shell sort has received much attention over many years, no one has been able to analyze it yet! It has been established that for many sequences of values of *h* (including those used in the code above), Shell sort never does more than $n^{1.5}$ comparisons in the worst case. Empirical studies have shown that it is quite fast on most lists. Hence Shell sort is the fastest sorting algorithm we have considered so far.

13.6 Summary and Conclusion

For small lists of less than a few hundred elements, any of the algorithms we have considered in this chapter are adequate. For larger lists, Shell sort is usually the best choice, except in a few special cases:

- If a list is nearly sorted, use insertion sort;
- If a list contains large records that are very expensive to move, use selection sort because it does the fewest number of data moves (of course, the fast algorithms we study in a later chapter are even better).

Never use bubble sort: it makes as many comparisons as any other sort, and usually moves more data than any other sort, so it is generally the slowest of all.

13.7 Review Questions

1. What is the difference between internal and external sorts?
2. The complexity of bubble sort and selection sort is exactly the same. Does this mean that there is no reason to prefer one over the other?
3. Does Shell sort have a best case that is different from its worst case?

13.8 Exercises

1. Rewrite the bubble sort algorithm to incorporate a check to see whether the array is sorted after each pass, and to stop processing when this occurs.
2. An alternative to bubble sort is the Cocktail Shaker sort, which uses swaps to move the largest value to the top of the unsorted portion, then the smallest value to the bottom of the unsorted portion, then the largest value to the top of the unsorted portion, and so forth, until the array is sorted.
   a) Write code for the Cocktail Shaker sort.
   b) What is the complexity of the Cocktail Shaker sort?
   c) Does the Cocktail Shaker sort have anything to recommend it (besides its name)?
3. Adjust the selection sort algorithm presented above to sort using the maximum rather than the minimum element in the unsorted portion of the array.
4. Every list of length *n* is *n*-nearly sorted. Using the formula for the worst case complexity of insertion sort on a *k*-nearly sorted list with $k = n$, we get $W(n) = n(n\text{-}1)$. Why is this result different from $W(n) = n(n\text{-}1)/2$, which we calculated elsewhere?

5. Shell sort is a modified insertion sort and insertion sort is very fast for nearly sorted lists. Do you think that Shell sort would sort nearly sorted lists even faster than insertion sort? Explain why or why not.
6. The sorting algorithms presented in this chapter are written for ease of analysis and do not take advantage of all the features of Ruby. Rewrite the sorting algorithms using as many features of Ruby as possible to shorten the algorithms or make them faster.
7. A certain data collection program collects data from seven remote stations that it contacts over the Internet. Every minute, the program sends a message to the remote stations prompting each of them to collect and return a data sample. Each sample is time stamped by the remote stations. Because of transmission delays, the seven samples do not arrive at the data collection program in time stamp order. The data collection program stores the samples in an array in the order in which it receives them. Every 24 hours, the program sorts the samples by time stamp and stores them in a database. Which sorting algorithm should the program use to sort samples before they are stored: bubble, selection, insertion, or Shell sort? Why?

13.9 Review Question Answers

1. An internal list processes lists stored in main memory, while an external sorts processes lists stored in files.
2. Although the complexity of bubble sort and selection sort is exactly the same, in practice they behave differently. bubble sort tends to be significantly slower than selection sort, especially when list elements are large entities, because bubble sort moves elements into place in the list by swapping them one location at a time while selection sort merely swaps one element into place on each pass. bubble sort makes $O(n^2)$ swaps on average, while selection sort $O(n)$ swaps in all cases. Had we chosen swaps as a basic operation, this difference would have been reflected in our analysis.
3. Shell sort does different things when the data in the list is different so it has best, worst, and average case behaviors that differ from one another. For example, it will clearly do the least amount of work when the list is already sorted, as is the case for insertion sort.