Chapter 4

Sorting

We have already looked at several sorting algorithms. In this chapter, however, we will look at this topic more formally.

In previous chapters, we applied our sorting algorithms to arrays of integers. Although this is a good way to introduce the algorithms, it is not very common in practical situations to sort an array of simple numbers. Usually, the items being sorted are more complex, typically objects or structs, so that each item is itself made up of a number of elements. We will refer to the items that are being sorted as **records** and to the elements that make up the items as **fields** of the records. For example, if each record is a name and address, then the fields might be first name, last name, street address, city, state, and zip code. When an array of records is to be sorted, the sorted order typically depends on only one of the fields of the record. This field is said to be the **key** field for the sort, and the values of this field are called keys. In other words, when we sort an array of records, we compare keys, and we arrange the records so that their keys are in ascending (or descending) order.

For example, we might want to sort an array of names and addresses by zip code. The zip code of each record would then be its key. Occasionally, a key might be made up of two or more fields of a record. For example, we might want to sort the names and addresses using a key that is a combination of last name and first name, so that the last names are in alphabetical order and within a group of people with the same last name, the first names are in alphabetical order.

In a key-based sorting procedures, only the keys are compared. As an example, let's look at Insertion Sort applied to an array of names and addresses. Assume that *MailingLabel* is a class that includes a field named *zipcode* of type string. We can apply Insertion Sort to sort an array of n *MailingLabel* objects by zip code:

```
1. void insertion_sort_by_zip( MailingLabel list[], int n ) {
2.    for ( int next = 1; next < n; next++ ) {
3.       int value = list[next];
4.       int pos = next;
5.       while ( pos > 0 && list[pos-1].zipcode > value.zipcode ) {
6.          list[pos] = list[pos-1];
```

```
 7.          pos--;
 8.       }
 9.       list[pos] = value;
10.    }
11. }
```

Compare this to Insertion Sort for an array of integers in Chapter 1. The only difference is that we compare key values rather than entire array elements.

It might be worth noting that a sorting procedure does a lot of copying of array elements. Copying a single integer does not take a lot of time, but copying a record that contains many fields can be very expensive. In Java, the values stored in the array are actually pointers to objects. When an array element is copied, only the pointer is copied; the object itself is not. In C++, an array of objects contains the actual objects, and copying an array element can be relatively time-consuming. In C++, it is of course possible to use an array of pointers to objects instead of an array of objects. It can be a good idea to do so when sorting large objects.

So far, we have been concerned with only the time complexity of sorting algorithms. We have seen that selection sort and insertion sort have worst-case run times that are $\Theta(n^2)$, while the worst-case run time of heap sort is $\Theta(n \log(n))$. However, it is also possible to consider space complexity. For sorting algorithms, the natural question is, How much *extra* space—beyond the space occupied by the array itself—is required for sorting an array of n elements. All the sorting algorithms that we have seen so far use a constant amount of extra space, independent of the size of the array. The only extra space required by these algorithms is for a few local variables in the sorting routines. This is not something that is true for every sorting algorithm: A variation of heap sort that puts items removed from the heap into a new separate array would require memory for the extra array. The amount of extra memory would be $\Theta(n)$.

Definition 4.1. A sorting algorithm is said to be an **in-place** sorting algorithm if the amount of extra space required by the algorithm is $\Theta(1)$. That is, the amount of extra space is bounded by a constant, independent of the size of the array.

HeapSort is an interesting sorting algorithm not just because it is a $\Theta(n \log(n))$ algorithm, but because it is an *in-place* $\Theta(n \log(n))$ sorting algorithm.

There is one more general property of sorting algorithms that we should consider. Suppose that an array of name/address records has been sorted into alphabetical order by name. Now, suppose that we apply some sorting algorithm to the array to sort it according to zip code. Consider a group of records that all have the same zip code. These records form a contiguous group in the array. The question is, are the records in this group still sorted into alphabetical order by name? That is, when we sorted according to zip code, did we disturb the relative ordering of items that have the same name? The answer is that it depends on the sorting algorithm that we use.

Definition 4.2. A sorting algorithm is said to be **stable** if items that have the same key remain in the same relative order after the sorting algorithm is applied as they were in before the sort.

Insertion Sort is a stable sorting algorithm because it never changes the ordering of items with the same key. In Selection Sort, on the other hand, an item can jump from one part of the array

to another, moving past other items with the same key. Selection Sort is not stable. Similarly, HeapSort is not a stable sorting algorithm.[1]

Stability is not always needed in a sorting algorithm, but it is desirable in some cases. Suppose, for example, we are sorting records that contain first name and last name fields. Suppose that we sort the array using the first name as the key, and then we sort using the last name as key. If we use a stable sorting algorithm, then records with the same last name will still be in alphabetical order by first name. If the algorithm is not stable, then people with the same last name can end up in an arbitrary order.

In the remainder of this chapter, we will look at several new sorting algorithms, starting with QuickSort, which is probably the most commonly used general sorting algorithm in practice.

4.1 QuickSort

Fundamentally, QuickSort is based on a simple idea: Pick some key. Put all the records that have a smaller key than the selected key at the beginning of the array, and put all the records with larger keys at the end. Then apply the same procedure recursively to each group of records, continuing until you get down to groups of size zero or one. However, organizing all this and doing it efficiently requires some cleverness.

A basic operation in QuickSort is the algorithm that I call *QuickSortStep*, which applies the procedure described in the preceding paragraph to the records between two specified positions in the array. The first record in the sequence is used as a "pivot." Records with smaller keys than the pivot are moved to the beginning of the sequence; records with larger keys are moved to the end. The pivot element itself ends up between the two groups of records. Note that records within one of the groups are *not* yet sorted, but the pivot element is in the final position that it will occupy when the entire sequence has been sorted. For an array of integers, QuickSortStep can be coded as follows:

```
 1.  int QuickSortStep( int list[], int low, int high ) {
 2.        // Apply QuickSortStep to the sequence list[low],
 3.        // list[low+1], ..., list[high].  The return value
 4.        // is the final position of the pivot element.
 5.      int pivot = list[low];
 6.      while (true) {
 7.         while (high > low && list[high] >= pivot)
 8.            high--;
 9.         if (high == low)
10.            break;
11.         list[low] = list[high];
12.         low++;
```

[1]The algorithm known as *Merge Sort* is a stable $\Theta(n \log(n))$ sorting algorithm. However, Merge Sort is not an in-place sorting algorithm. It uses an auxiliary array of the same size as the original array and so has space complexity $\Theta(n)$.

```
13.         while (high > low && list[low] <= pivot)
14.            low++;
15.         if (high == low)
16.            break;
17.         list[high] = list[low];
18.         high--;
19.      }
20.      list[low] = pivot;
21.      return low;
22.   }
```

In this algorithm, *pivot* holds the pivot value. As the values of *low* and *high* change, positions below *low* hold items that are known to be less than or equal to the pivot, and positions above *high* hold items that are known to be greater than or equal to the pivot. Positions between *low* and *high* hold items that are unclassified. *Low* and *high* gradually move towards each other. When they become equal, there are no more unclassified items and the mutual value of *low* and *high* represents the final position of the pivot, between the two groups of items. This final position is returned by the function, since it is needed in the full QuickSort algorithm.

You should think of line 5 as leaving an empty space in the array at position *low*. Lines 7 and 8 look for an item from the top end of the sequence that is less than the pivot and therefore can be moved to position *low*. When this item is moved, in line 11, there is an empty space in position *high*. Lines 13 and 14 look for an item greater than the pivot to fill this space. When this item is moved in line 17, the empty space is once again in position *low*. We break out of the loop as soon as *high* becomes equal to *low*. In line 20, the pivot is placed into the final position of the empty space.

With a QuickSortStep function in hand, it is easy to write a QuickSort function. The function is recursive and is written so as to apply to a specified range of positions in the array. If A is an array of n integers, we would sort the entire array by calling $QuickSort(A,0,n-1)$.

```
1.  void QuickSort(int list[], int low, int high) {
2.        // Sort the elements list[low], list[low+1], ...,
3.        // list[high] using QuickSort.
4.     if (high > low) {
5.        int mid = QuickSortStep(list,low,high);
6.        QuickSort(list,low,mid-1);
7.        QuickSort(list,mid+1,high);
8.     }
9.  }
```

The base case of the recursion occurs when $high \leq low$. In this base case, the array segment that is being sorted has fewer than two elements, so there is nothing to do. In the recursive case, we start by applying QuickSortStep to the array segment. The variable *mid* is the position of the pivot element that was used by QuickSortStep. This element is in its correct final location. We can

then complete the sort by sorting the elements in locations *low* through $mid - 1$, which precede the pivot, and the elements in positions $mid + 1$ through *high*, which follow the pivot. Note that either of these sub-segments might have length zero or one. In that case, the corresponding call to QuickSort will return without doing anything.

QuickSort is quite fast on average. Its average case run time is $\Theta(n \log(n))$, although this fact is not so easy to prove. We can note that the run time for QuickSortStep is proportional to the length of the array segment to which it is applied. When QuickSort is applied to an array of n elements, the call to QuickSortStep on the top level does work proportional to n. On the next level, QuickSortStep is applied to two sub-arrays, but the combined length of these sub-arrays is just $n - 1$, so the *total* work done by the two calls to QuickSortStep on the second level of the recursion is still bounded by a constant times n. Three levels down the recursion, QuickSortStep can be applied to as many as four sub-arrays, but the combined length of all the sub-arrays is less than n and so the total work done by all calls to QuickSortStep on the third level of the recursion is also bounded by a constant times n. In fact, the work done on *every* level of the recursion is bounded by a constant times n. If the recursion continues k levels deep, the total work done is bounded by a constant times $n * k$. Now, if QuickSortStep always divided the sub-array into two equal parts, there would be $\log(n)$ levels, and the run time would be bounded by a constant times $n * \log(n)$. Thus, the run time would be $\Theta(n \log(n))$. QuickSortStep does not usually divide the sub-array exactly in half. But it comes close enough on average that the average run time is still $\Theta(n \log(n))$.

In practice, QuickSort is significantly faster than HeapSort on average (by a constant multiple). However, the worst-case run time for QuickSort is $\Theta(n^2)$. This worst case occurs, curiously, when QuickSort is applied to an array that is already in sorted order. In this case, the pivot element chosen by QuickSortStep is already in its final position, so that QuickSortStep will divide a sub-array of length m into an array of length 0 and an array of length $m - 1$. For an array of length n, QuickSortStep is applied to n elements on the top level of the recursion. On the second level, it is applied to one sub-array with $n - 1$ elements. On the third level, it is applied to $n - 2$ elements. And so on. There are n levels to this recursion, rather than $\log(n)$, and the total amount of work done is proportional to n^2. For randomly ordered arrays, run times close to the worst case are very rare. However, if QuickSort is applied to arrays that are sorted or almost sorted, the run time can be a problem.

This problem is sometimes addressed by using *Randomized QuickSortStep*. In this version, instead of choosing the first element of the sub-array as the pivot, the pivot element is chosen randomly from the sub-array. This random element is swapped with the first element, and then Randomized QuickSortStep proceeds in the same way as the regular version. This does not change the worst-case run time of QuickSort, but it does mean that the worst case is extremely unlikely to occur for almost-sorted arrays. Since almost-sorted arrays are common in some application, Randomized QuickSort is often preferred over regular QuickSort as a general purpose sorting algorithm.

Because of the way QuickSortStep jumps elements from one part of the array to another, QuickSort is not a stable sorting algorithm. It is also not in-place. This is clear if we note that each recursive call in the QuickSort algorithm requires space on the stack of activation records that is used to implement recursion. This stack memory is part of the extra space that is required by

QuickSort. The amount of stack memory needed is proportional to the depth of the recursion. This depth can be as little as $\log(n)$, but can be as much as n. On the average, it is much closer to $\log(n)$ than to n. So, the average-case extra memory requirement of QuickSort is $\Theta(\log(n))$, but the worst case is $\Theta(n)$. However, by the end of the next section, we'll see a variation of QuickSort for which the worst-case extra space requirement is $\Theta(\log(n))$.

4.2 Stacks and Recursion

Recursive subroutines are implemented using a stack of activation records. This is done automatically, and a programmer who wants to use recursion does not have to think about stacks. However, it is possible for a programmer to implement a recursive algorithm "by hand," using a stack data structure and a non-recursive routine. Since calling a function can be a relatively expensive operation, it is possible that a non-recursive routine that uses a stack might be faster than a recursive routine that implements the same algorithm.

In recursion, a problem is broken down into subproblems. Since it's only possible to work on one subproblem at a time, some of the subproblems must be put aside for future processing. They are placed on a stack. When one subproblem is finished, the next one is removed from the stack for processing. We can develop this idea into a general pseudocode algorithm for using a stack to solve a problem recursively:

```
Push the main problem onto the stack
While the stack is not empty {
   Pop a problem from the stack
   If the problem is a base case {
      Solve the problem directly
   }
   Else {
      Break the problem into subproblems
      Push each subproblem onto the stack
   }
}
```

In the case of QuickSort, a "problem" is a sub-array that we want to sort. QuickSortStep is used to break this problem into two subproblems, which are then solved separately. We can represent each subproblem by the two integers that mark the beginning and end of the sub-array. We can store subproblems using a simple array of integers, as long as we put two integers onto the array for each problem. That is, pushing a problem will mean pushing two integers and popping a problem will mean popping two integers. Here is a version of QuickSort that uses this idea:

```
void QuickSort(int list[], int n) {
     // Sort an array of n integers
  IntStack stack;  // A stack to hold subproblems
```

```
  stack.push(n-1);  // Push the main problem onto the stack
  stack.push(0);

  while ( ! stack.isEmpty() ) {

     int low = stack.pop();   // Get a problem from the stack.
     int high = stack.pop();

     if (high <= low) {
          // base case, nothing to do
     }
     else {
        int mid = QuickSortStep(list,low,high);

        stack.push(mid-1);  // Push first subproblem onto stack.
        stack.push(low);

        stack.push(high);   // Push second subproblem onto stack.
        stack.push(mid+1);
     }

  } // end while
}
```

We can improve on this a bit by noting that we keep pushing problems onto the stack that are only going to be popped off immediately. We don't really need to stack the problem that we are going to work on next. With this idea, our pseudocode algorithm becomes:

```
Let P be the main problem
While (true) {
   If P is a base case {
      Solve P directly
      If the stack is empty
         break;
      Pop a problem from the stack.
   }
   Else {
      Break P into subproblems S1, S2, S3...
      Push S2, S3, ... onto the stack
      Let P be S1
   }
}
```

For QuickSort, we will only push one subproblem onto the stack and work on the other one. In fact, the algorithm that we use always pushes the *longer* of the two subproblems onto the stack and works on the shorter problem. Note that the shorter subproblem is always less than half the size of the original problem. The result of this is that the stack never grows larger than $2 * \log(n)$, where n is the size of the entire array. That is, we have an algorithm that has worst-case extra space requirement $\Theta(\log(n))$. Here is the code for this version of QuickSort:

```
void QuickSort(int list[], int n) {
  IntStack stack;  // A stack to hold subproblems

  int low = 0;     // The main problem
  int high = n-1;

  while ( true ) {

     if (high <= low) {
          // base case, nothing to do for this problem.
        if (stack.isEmpty())
           break;
        low = stack.pop();  // Pop a problem from the stack.
        high = stack.pop();
     }
     else {
        int mid = QuickSortStep(list,low,high);
        if ( (high-mid) > (mid-low) ) {

           stack.push(high);   // Push longer subproblem onto stack.
           stack.push(mid+1);

           high = mid - 1;  // Work on the shorter problem;
                            // The value of low stays the same.
        }
        else {

           stack.push(mid-1);   // Push longer subproblem onto stack.
           stack.push(low);

           low = mid + 1;   // Work on the shorter problem;
                            // The value of high stays the same.
        }
     }
  } // end while
}
```

4.3 Radix Sort

So far, every sorting algorithm that we have looked at has worst-case run time that is $\Theta(n \log(n))$ or greater. The same is true for the average case. This is no accident. For a large class of sorting algorithms, it can be shown that $\Theta(n \log(n))$ is the best we can do for the worst and average case run times. The type of algorithm in question is called a **comparison sort**. A comparison sort can compare two keys to find out which one is greater, but it cannot look at the internal structure of a key. If we lift this restriction and allow a sorting algorithm to inspect the structure of a key, we can find non-comparison sorts that have worst case run time $\Theta(n)$. One of these is *Radix Sort*.

It is easiest to think of Radix Sort as working on lists, rather than on arrays. Let's see how Radix Sort would be applied to sort a list of k-digit positive integers. The integers might be zip codes or social security numbers, for example. Radix Sort first separates the integers into 10 lists, based on the last (least significant) digit. Numbers in which the last digit is 0 go into the first list, numbers in which it is 1 go into the second list, and so one. The ten lists are the concatenated back into one list. In this list, the numbers are sorted into order according to the value of their least significant digits. This operation can be performed with a run time that is proportional to the length of the list.

Now, repeat the same operation, but using the next-to-last digit. And continue to do the same for each of the digits in the numbers. In the end, the numbers will be completely sorted. We have done k passes through the numbers, where k is the number of digits. Each pass has run time proportional to n. Since k is a constant, the total run time is also proportional to n. As long as we limit ourselves to keys with a fixed maximum number of digits, Radix Sort is a $\Theta(n)$ sorting algorithm. Note however that this is mostly of theoretical interest, since for arrays of any reasonable size, QuickSort is faster than Radix Sort.

Radix Sort is also of historical interest, since it is the method that was used by machines that sorted punched cards according to the value of some number punched onto the cards. The machine would separate the cards into bins based on the value of some specified digit in the number. The cards would first be sorted into bins according to their least significant digits. Then the cards would be removed from the bins, stacked up, and fed through the machine again to sort them according to the next digit. After a sufficient number of passes through the machine, the cards would be completely sorted.

We can easily write a RadixSort routine that uses the *IntList* ADT from the previous chapter to sort an array of positive integers. We use ten lists, one for each possible digit. I also assume that we have a function *digit*(n,d) that returns the d^{th} digit of n for a positive integer n.

```
void RadixSort(int list[], int n, int digits) {
       // Sort an array of n positive integers, where the
       // maximum number of digits in any of the integers
       // is given by the third parameter.

   IntList digitlist[10];  // One list for each possible digit.

   for (int digitnum = 0; digitnum < digits; digitnum++) {
```

```
            for (int i = 0; i < n; i++) {
                   // Place item i into one of the 10 lists, based
                   // on the digit at position digitinum in that item.
               int d = digit(list[i],digitnum);  // Digit from item i.
               digitlist[d].insert( digitlist[d].end(), list[i] );
            }

            int ct = 0;  // Number of items retrieved from the lists
            for (int d = 0; d < 10; d++) {
                  // Move the items from list number d back to the array.
               while (digitlist[d].begin() != digitlist[d].end()) {
                  list[ct] = digitlist[d].get(digitlist[d].begin());
                  ct++;
                  digitlist[d].remove(digitlist[d].begin());
               }
            }

         }
      }
```

Exercises

1. Do an experiment to determine how much slower the recursive version of QuickSort is than the version that uses a stack (if at all).
2. For very short arrays, Insertion Sort is actually faster than QuickSort. Write a version of QuickSort that uses Insertion Sort to sort a sub-array whose length is less than some specified limit. Do an experiment to test the performance of this version of QuickSort for various array size limits.
3. Is Radix Sort an in-place sort? Is it stable?