# Basic Sorting Algorithms

Two of the most common operations performed on data stored in a computer are sorting and searching. This has been true since the beginning of the computer industry, so this means that sorting and searching are two of the most studied operations in computer science. Many of the data structures discussed in this book are designed primarily to make sorting and/or searching the data stored in the data structure easier and more efficient.

This chapter introduces you to some of the basic algorithms for sorting data. These algorithms depend only on the array as the means of storing data. In this chapter we'll also look at ways of timing our programs to determine which of the algorithms is most efficient.

## Sorting Algorithms

Much of the data we work with in our day-to-day lives is sorted. We look up the definitions for words in a dictionary by searching the word list alphabetically. We look up a phone number in our contacts list by searching through names alphabetically (I almost used a phone book for the example). The post office sorts mail by working through the zip code and then the street address. This chapter introduces you to several different techniques for sorting data. Please keep in my mind while reading this chapter is that these sorts work best on small amounts of data. In a later chapter we will discuss sorting algorithms for large data sets.

## An Array Test Bed

To examine these basic sorting algorithms, we will first need a test bed in which to implement and test them. We'll build a class for array data that encapsulates some of the normal array operations: inserting new data, displaying the array data, and calling the different sorting algorithms.

Here is the code for our array class:

function CArray(numElements) {

this.dataStore = [];

this.pos = 0;

this.numElements = numElements;

this.insert = insert;

this.toString = toString;

this.clear = clear;

this.setData = setData;

for (var i = 0; i < numElements; ++i) {

this.dataStore[i] = i;

}

}

function setData() {

for (var i = 0; i < this.numElements; ++i) {

this.dataStore[i] = Math.floor(Math.random() \*

(this.numElements+1));

}

}

function clear() {

for (var i = 0; i < this.dataStore.length; ++i) {

this.dataStore[i] = 0;

}

}

function insert(element) {

this.dataStore[this.pos++] = element;

}

function toString() {

var retstr = "";

for (var i = 0; i < this.dataStore.length; ++i) {

retstr += this.dataStore[i] + " ";

if (i > 0 && i % 10 == 0) {

retstr += "\n";

}

}

return retstr;

}

Here is a simple program that uses the CArray class (the class is named CArray because JavaScript already has an Array class):

var numElements = 100;

var mynums = new CArray(numElements);

print(mynums.toString());

The output from this program is:

0 1 2 3 4 5 6 7 8 9 10

11 12 13 14 15 16 17 18 19 20

21 22 23 24 25 26 27 28 29 30

31 32 33 34 35 36 37 38 39 40

41 42 43 44 45 46 47 48 49 50

51 52 53 54 55 56 57 58 59 60

61 62 63 64 65 66 67 68 69 70

71 72 73 74 75 76 77 78 79 80

81 82 83 84 85 86 87 88 89 90

91 92 93 94 95 96 97 98 99

### Generating Random Data

Creating an array is easy to do with a for loop, but it is a little harder to get the data in some type of unsorted form. To create an array of random data, we need to generate random numbers for our array class data. JavaScript has a method for generating random numbers that is part of the Math library – Math.random(). This method will generate a random floating point number between 0.0 and 1.0. To demonstrate, look at the following short program:

for (var i = 1; i < 10; ++i) {

print(Math.random());

}

This program will produce the following list of numbers:

0.7101997691692159

0.7223845682126121

0.9862072533160854

0.41110121147275525

0.8309579288861452

0.13745391563440057

0.2673191050553345

0.4031226500657903

0.2962118116798562

These numbers are not very useful for exploring the outcome of a sorting algorithm, so we need to modify them. We can specify a range by multiplying the generated number by a number that is one more than the upper limit of the range we want. For example, if I want a set of random numbers in the range from 0 to 10, I can modify the above program like this:

for (var i = 1; i < 10; ++i) {

print(Math.random() \* 11);

}

The resulting output is:

1.310519669590743

5.765661793996499

9.918426265757144

9.75431799802004

7.913001179119491

2.8169587051198155

1.8689941424790997

10.526037246333603

4.917621833509056

This set of numbers is better, but still too hard to read. We need to make one more adjustment, by calling the Math.floor() method to round the number down:

for (var i = 1; i < 10; ++i) {

print(Math.floor(Math.random() \* 11));

}

Now we can apply this technique to generating our array data set:

for (var i = 0; i < numElements; ++i) {

this.dataStore[i] = Math.floor(Math.random() \* (numElements+1));

}

After we call the constructor now to create a new array we get the following data set:

76 69 64 4 64 73 47 34 65 93 32

59 4 92 84 55 30 52 64 38 74

40 68 71 25 84 5 57 7 6 40

45 69 34 73 87 63 15 96 91 96

88 24 58 78 18 97 22 48 6 45

68 65 40 50 31 80 7 39 72 84

72 22 66 84 14 58 11 42 7 72

87 39 79 18 18 9 84 18 45 50

43 90 87 62 65 97 97 21 96 39

7 79 68 35 39 89 43 86 5

The JavaScript random number generator is not perfect for several reasons, including not being able to specify a seed value, but it will do for the purposes of allowing us to generate data random enough to be interesting to sort.

## Bubble Sort

The first sorting algorithm we'll examine is the bubble sort. The bubble sort is one of the slowest sorting algorithms, but it is also one of the easiest sorts to implement.

The bubble sort gets its name because when data are sorted using the algorithm, values float like a bubble from one end of the array to the other. Assuming you are sorting a set of numbers into ascending order, larger values float to the right and lower values float to the left. This behavior is the result of the algorithm moving through the array many times, comparing adjacent values, and swapping them if the value to the left is greater than the value to the right.

Figure x.1 illustrates how the bubble sort works. In the figure, we examine two particular values inserted into the array, 2 and 72. Each number is marked with a circle around it. You can watch how 72 moves from the beginning of the array to the middle of the array, and you can watch how 2 moves from just past the middle of the array to the beginning of the array.

(insert Figure x.1 here)

Here is the code for the bubble sort:

function bubblesort() {

var numElements = this.dataStore.length;

var temp;

for (var outer = numElements; outer >= 2; --outer) {

for (var inner = 0; inner <= outer-1; ++inner) {

if (this.dataStore[inner] > this.dataStore[inner+1]) {

temp = this.dataStore[inner];

this.dataStore[inner] = this.dataStore[inner+1];

this.dataStore[inner+1] = temp;

}

}

}

}

Here is a short program that sorts ten numbers using the bubblesort() method:

var numElements = 10;

var mynums = new CArray(numElements);

print(mynums.toString());

mynums.bubblesort();

print();

print(mynums.toString());

The output from this program is:

10 8 3 2 2 4 9 5 4 3

2 2 3 3 4 4 5 8 9 10

We can see that the bubble sort algorithm works, but it would be nice to view the intermediate results of the algorithm. We can do that by the careful placement of the toString() method into the bubblesort() method:

function bubblesort() {

var numElements = this.dataStore.length;

var temp;

for (var outer = numElements; outer >= 2; --outer) {

for (var inner = 0; inner <= outer-1; ++inner) {

if (this.dataStore[inner] > this.dataStore[inner+1]) {

temp = this.dataStore[inner];

this.dataStore[inner] = this.dataStore[inner+1];

this.dataStore[inner+1] = temp;

}

}

print(this.toString());

}

}

When we rerun the program above with the new method, we get the following output:

1 0 3 3 5 4 5 0 6 7

0 1 3 3 4 5 0 5 6 7

0 1 3 3 4 0 5 5 6 7

0 1 3 3 0 4 5 5 6 7

0 1 3 0 3 4 5 5 6 7

0 1 0 3 3 4 5 5 6 7

0 0 1 3 3 4 5 5 6 7

0 0 1 3 3 4 5 5 6 7

0 0 1 3 3 4 5 5 6 7

0 0 1 3 3 4 5 5 6 7

0 0 1 3 3 4 5 5 6 7

With this output, you can easily view how the lower values work their way to the beginning of the array and the higher values work their way to the end of the array.

## Selection Sort

The next sorting algorithm to examine is the selection sort. This sort works by starting at the beginning of an array, comparing the first element with the remaining elements in the array. After examining all the elements, the smallest element is placed in the first position of the array, and the algorithm moves to the second position. This process continues until the algorithm arrives at the next to last position in the array.

Nested loops are used in the selection sort algorithm. The outer loop moves from the first element in the array to the next to last element; the inner loop moves from the second array element to the last element, looking for values that are smaller than the element currently being pointed to by the outer loop. After each iteration of the inner loop, the smallest value in the array is assigned its proper place in the array. Figure x.2 illustrates how the selection sort algorithm works.

(Insert Figure x.2 here)

Here is the code for the selectionsort() method:

function selectionsort() {

var min, temp;

for (var outer = 0; outer <= this.dataStore.length-2; ++outer) {

min = outer;

for (var inner = outer + 1;

inner <= this.dataStore.length-1; ++inner) {

if (this.dataStore[inner] < this.dataStore[min]) {

min = inner;

}

}

temp = this.dataStore[outer];

this.dataStore[outer] = this.dataStore[min];

this.dataStore[min] = temp;

}

}

Here is the output from a run of our program using the selectionsort() method (add the line print(this.toString(); right after the swap:

6 8 0 6 7 4 3 1 5 10

0 8 6 6 7 4 3 1 5 10

0 1 6 6 7 4 3 8 5 10

0 1 3 6 7 4 6 8 5 10

0 1 3 4 7 6 6 8 5 10

0 1 3 4 5 6 6 8 7 10

0 1 3 4 5 6 6 8 7 10

0 1 3 4 5 6 6 8 7 10

0 1 3 4 5 6 6 7 8 10

0 1 3 4 5 6 6 7 8 10

0 1 3 4 5 6 6 7 8 10

## Insertion Sort

The insertion sort is analogous to the way humans sort things numerically or alphabetically. Let's say I have asked a class of students to turn in an index card with his or her name, identification number, and a short biographical sketch. The students return the cards in random order, but I want them to be alphabetized so that I can build a seating chart.

I take the cards back to my office, clear off my desk, and pick the first card. The last name on the card is Smith. I place it at the top left position of the desk and pick the second card. The last name on the card is Brown. I move Smith over to the right and put Brown in Smith's place. The next card is Williams. It can be inserted at the far right without having to shift any other cards. The next card is Acklin. It has to go at the beginning of the list, so each of the other cards must be shifted one position to the right to make room for the Acklin card. This is how the insertion sort works.

Here is the code for the insertion sort:

function insertionsort() {

var temp, inner;

for (var outer = 1; outer <= this.dataStore.length-1; ++outer) {

temp = this.dataStore[outer];

inner = outer;

while (inner > 0 && (this.dataStore[inner-1] >= temp)) {

this.dataStore[inner] = this.dataStore[inner-1];

--inner;

}

this.dataStore[inner] = temp;

}

}

The insertion sort has two loops. The outer loop moves element-by-element through the array, while the inner loop compares the element chosen in the outer loop to the element next to it in the array. If the element selected by the outer loop is less than the element selected by the inner loop, array elements are shifted over to the right to make room for the inner loop element, just as described in the example above.

Now let's look at how the insertion sort works by running our program with a data set:

6 10 0 6 5 8 7 4 2 7

0 6 10 6 5 8 7 4 2 7

0 6 6 10 5 8 7 4 2 7

0 5 6 6 10 8 7 4 2 7

0 5 6 6 8 10 7 4 2 7

0 5 6 6 7 8 10 4 2 7

0 4 5 6 6 7 8 10 2 7

0 2 4 5 6 6 7 8 10 7

0 2 4 5 6 6 7 7 8 10

0 2 4 5 6 6 7 7 8 10

This output clearly shows that the insertion sort works not by making data exchanges, but by moving larger array elements to the right to make room for the smaller elements on the left side of the array.

## Timing Comparisons of the Basic Sorting Algorithms

These three sorting algorithms are very similar in complexity, and theoretically, they should perform similarly. To determine the differences in performance among these three algorithms, we can use an informal timing system to compare the times it takes them to sort data sets.

The timing system we will use in this section is based on the getting the system time using the JavaScript Date object's getTime() method. Here is how the method works:

var start = new Date().getTime();

The getTime() method returns the system time in milliseconds. The following code fragment:

var start = new Date().getTime();

print(start);

results in the following output:

1351548727209

To record the time it takes code to execute, we need to also capture the stop time, with the executing code going between the two time captures. Here is an example of timing a for loop that displays the numbers 1-100:

var start = new Date().getTime();

for (var i = 1; i < 100; ++i) {

print(i);

}

var stop = new Date().getTime();

var elapsed = stop - start;

print("The elapsed time was: " + elapsed + " milliseconds.");

The output, not including each number, from the program is:

The elapsed time was: 91 milliseconds.

Now that we have a tool for measuring the efficiency of these sorting algorithms, let's run some tests to compare them.

### Comparing the Basic Sorting Algorithms

For our comparison of the three basic sorting algorithms, we will time the three algorithms sorting arrays with data set sizes of 100, 1000, and 10000. We expect to not see much difference among the algorithms for data set sizes of 100 and 1000, but we do expect there to be some difference when using a data set size of 10000.

Let's start with an array of 100 randomly chosen integers. We also added a function for creating a new test bed for each algorithm. Here's the code for the test:

function setData(arr, numElements) {

for (var i = 0; i < numElements; ++i) {

arr.insert(Math.floor(Math.random() \* (numElements+1)));

}

}

var numElements = 100;

var nums = new CArray(numElements);

var start = new Date().getTime();

nums.bubblesort();

var stop = new Date().getTime();

var elapsed = stop - start;

print("Elapsed time for the bubble sort on " + numElements +

" elements is: " + elapsed + " milliseconds.");

setData();

start = new Date().getTime();

nums.selectionsort();

stop = new Date().getTime();

elapsed = stop - start;

print("Elapsed time for the selection sort on " + numElements +

" elements is: " + elapsed + " milliseconds.");

setData();

start = new Date().getTime();

nums.insertionsort();

stop = new Date().getTime();

elapsed = stop - start;

print("Elapsed time for the insertion sort on " + numElements +

" elements is: " + elapsed + " milliseconds.");

Here are the results:

Elapsed time for the bubble sort on 100 elements is: 0 milliseconds.

Elapsed time for the selection sort on 100 elements is: 1 milliseconds.

Elapsed time for the insertion sort on 100 elements is: 0 milliseconds.

Clearly, there is not any significant difference among the three algorithms.

For the next test, we change the numElements variable to 1000. Here are the results:

Elapsed time for the bubble sort on 1000 elements is: 12 milliseconds.

Elapsed time for the selection sort on 1000 elements is: 7 milliseconds.

Elapsed time for the insertion sort on 1000 elements is: 6 milliseconds.

For 1000 numbers, the selection sort and the insertion sort are almost twice as fast as the bubble sort.

Finally, we test the algorithms with 10000 numbers:

Elapsed time for the bubble sort on 10000 elements is: 1096 milliseconds.

Elapsed time for the selection sort on 10000 elements is: 591 milliseconds.

Elapsed time for the insertion sort on 10000 elements is: 471 milliseconds.

The results for 10000 numbers are consistent with the results for 1000 numbers. Selection sort and insertion sort are faster than the bubble sort and the insertion sort is the fastest of the three sorting algorithms. Keep in mind, however, that these tests need to be run many, many times in order for the results to be considered statistically valid.

## Exercises

1. Run the three algorithms discussed in this chapter with string values rather than numeric values and compare the running times for the different algorithms. Are the results consistent with the results of using numeric data?
2. Create an array of 1000 integers already sorted in numeric order. Write a program that runs each sorting algorithm with this array, timing each algorithm, and comparing the times. How do these times compare to the times for sorting an array in random order?
3. Create an array of 1000 integers sorted in reverse numerical order. Write a program that runs each sorting algorithm with this array, timing each algorithm, and compare the times.