# Advanced Sorting Algorithms

In this chapter we examine algorithms for sorting data that are more complex than the algorithms examined in the chapter in Chapter x. These algorithms are also more efficient, and one of them, the *Quicksort* algorithm, is generally considered to be the most efficient sorting algorithm to use in most situations. The other sorting algorithms we'll examine in this chapter are the *Shellsort*, the *Mergesort*, and the *Heapsort*.

To compare these advanced sorting algorithms, we'll first discuss how each to implement each one, and in the exercises we will perform timing tests to compare their efficiency.

## Array Test Bed

Here is the test bed we will use for working with these sorting algorithms:

function CArray(numElements) {

this.dataStore = [];

this.pos = 0;

this.numElements = numElements;

this.insert = insert;

this.toString = toString;

this.clear = clear;

this.setData = setData;

this.setGaps = setGaps;

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;

}

## The Shellsort Algorithm

The Shellsort algorithm is named after its inventor, Donald Shell. This algorithm is a great improvement over the insertion sort. The algorithm's key concept is that it compares distant elements first, rather than adjacent elements, as is done in the insertion sort. Elements that are far out-of-place can be put into place more efficiently using this scheme than by simply comparing neighboring elements. As the algorithm loops through the data set, the distance between each element decreases until, when at the end of the data set, the algorithm is comparing elements that are adjacent.

Shellsort works by defining a gap sequence that indicates how far apart the algorithm will begin when comparing array elements. The gap sequence can be defined dynamically, but for most practical applications you can predefine the gap sequence the algorithm will use. There are several published gap sequences with different results. We are going to use the sequence defined by Marcin Ciura in his paper on best increments for average case of Shellsort (Ciura, 2001). This gap sequence is: 701,301,132,57,23,10,4,1. However, before we write our code for the average case, we are going to examine how the algorithm works with a small data set.

Let's start with a look at the code for the Shellsort algorithm:

function shellsort() {

for (var g = 0; g < this.gaps.length; ++g) {

for (var i = this.gaps[g]; i < this.dataStore.length; ++i) {

var temp = this.dataStore[i];

for (var j = i; j >= this.gaps[g] &&

this.dataStore[j-this.gaps[g]] > temp;

j -= this.gaps[g]) {

this.dataStore[j] = this.dataStore[j - this.gaps[g]];

}

this.dataStore[j] = temp;

}

}

}

For this program to work with our CArray class test bed, we also need to add a definition of the gap sequence. The following code goes into the constructor function for CArray:

this.gaps = [5,3,1];

The outer loop controls the movement within the gap sequence. In other words, for the first pass through the data set, the algorithm is going to examine elements that are 5 places away from each other. The next pass will examine elements that are 3 places away from each other. The last pass performs a standard insertion sort on elements that are 1 place away, or right next to each other. By the time this last pass begins, many of the elements will already be in place, and the algorithm won't have to exchange that many elements. This is where the efficiency of the algorithm is found. Figure x.1 illustrates how the Shellsort algorithm works on a data set of 10 random numbers where the gap sequence is: 5,3,1.

Now let's put the algorithm to work with a real example. We are going to add a print() statement to the shellsort() function so that we can follow the progress of the algorithm while it sorts the data set. Each gap pass is noted, followed by the order of the data set after sorting with that particular gap. Here is the complete program:

function CArray(numElements) {

this.dataStore = [];

//this.gaps = [701,301,132,57,23,10,4,1];

this.gaps = [5,3,1];

this.pos = 0;

this.numElements = numElements;

this.insert = insert;

this.toString = toString;

this.clear = clear;

this.setData = setData;

this.shellsort = shellsort;

this.setGaps = setGaps;

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

this.dataStore[i] = i;

}

}

function setGaps(arr) {

this.gaps = arr;

}

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;

}

function shellsort() {

for (var g = 0; g < this.gaps.length; ++g) {

for (var i = this.gaps[g]; i < this.dataStore.length; ++i) {

var temp = this.dataStore[i];

for (var j = i; j >= this.gaps[g] &&

this.dataStore[j-this.gaps[g]] > temp;

j -= this.gaps[g]) {

this.dataStore[j] = this.dataStore[j - this.gaps[g]];

}

this.dataStore[j] = temp;

}

print(this.toString());

}

}

var nums = new CArray(10);

nums.setData();

print("Before Shellsort: \n");

print(nums.toString());

print("\nDuring Shellsort: \n");

nums.shellsort();

print("\nAfter Shellsort: \n");

print(nums.toString());

The output from this program is:

Before Shellsort:

7 1 8 4 1 7 0 0 6 9

During Shellsort:

5 gap: 7 0 0 4 1 7 1 8 6 9

3 gap: 1 0 0 4 1 6 7 8 7 9

1 gap: 0 0 1 1 4 6 7 7 8 9

After Shellsort:

0 0 1 1 4 6 7 7 8 9

If you look at the data set after the 5 gap sequence was sorted, you will see that the number 7 is still at the beginning. That's because the algorithm compared it with the 5th next element in the data set, which is also the number 7, so no swap was performed.

Now having seen some details of how the Shellsort algorithm performs, let's use the larger gap sequence, take the print() statement out of the shellsort() function, and run it with a larger data set (100 elements). Here is the output:

Before Shellsort:

19 19 54 60 66 69 45 40 36 90 22

93 23 0 88 21 70 4 46 30 69

75 41 67 93 57 94 21 75 39 50

17 8 10 43 89 1 0 27 53 43

51 86 39 86 54 9 49 73 62 56

84 2 55 60 93 63 28 10 87 95

59 48 47 52 91 31 74 2 59 1

35 83 6 49 48 30 85 18 91 73

90 89 1 22 53 92 84 81 22 91

34 61 83 70 36 99 80 71 1

After Shellsort:

0 0 1 1 1 1 2 2 4 6 8

9 10 10 17 18 19 19 21 21 22

22 22 23 27 28 30 30 31 34 35

36 36 39 39 40 41 43 43 45 46

47 48 48 49 49 50 51 52 53 53

54 54 55 56 57 59 59 60 60 61

62 63 66 67 69 69 70 70 71 73

73 74 75 75 80 81 83 83 84 84

85 86 86 87 88 89 89 90 90 91

91 91 92 93 93 93 94 95 99

We will revisit this version of the algorithm again when we compare it to other sorting algorithms at the end of the chapter.

### Computing a Dynamic Gap Sequence

Sedgewick (Sedgewick, 2011) defines a shellsort() function that uses a formula to dynamically determine the gap sequence to use when running Shellsort. Sedgewick's algorithm determines the initial gap value using the following code fragment:

var N = this.dataStore.length;

var h = 1;

while (h < N/3) {

h = 3 \* h + 1;

}

Once this value is determined, the algorithm works like our previous Shellsort algorithm, except the last statement before going back into the outer loop computes a new gap value like so:

h = (h-1)/3;

Here is the definition of the complete function, along with a swap() function it uses:

function shellsort1() {

var N = this.dataStore.length;

var h = 1;

while (h < N/3) {

h = 3 \* h + 1;

}

while (h >= 1) {

for (var i = h; i < N; i++) {

for (var j = i; j >= h && this.dataStore[j] < this.dataStore[j-h];

j -= h) {

swap(this.dataStore, j, j-h);

}

}

h = (h-1)/3;

}

}

function swap(arr, index1, index2) {

var temp = arr[index1];

arr[index1] = arr[index2];

arr[index2] = temp;

}

Here is a program and its output that uses this new Shellsort algorithm:

var nums = new CArray(100);

nums.setData();

print("Before Shellsort1: \n");

print(nums.toString());

nums.shellsort1();

print("\nAfter Shellsort1: \n");

print(nums.toString());

Before Shellsort1:

92 31 5 96 44 88 34 57 44 72 20

83 73 8 42 82 97 35 60 9 26

14 77 51 21 57 54 16 97 100 55

24 86 70 38 91 54 82 76 78 35

22 11 34 13 37 16 48 83 61 2

5 1 6 85 100 16 43 74 21 96

44 90 55 78 33 55 12 52 88 13

64 69 85 83 73 43 63 1 90 86

29 96 39 63 41 99 26 94 19 12

84 86 34 8 100 87 93 81 31

After Shellsort1:

1 1 2 5 5 6 8 8 9 11 12

12 13 13 14 16 16 16 19 20 21

21 22 24 26 26 29 31 31 33 34

34 34 35 35 37 38 39 41 42 43

43 44 44 44 48 51 52 54 54 55

55 55 57 57 60 61 63 63 64 69

70 72 73 73 74 76 77 78 78 81

82 82 83 83 83 84 85 85 86 86

86 87 88 88 90 90 91 92 93 94

96 96 96 97 97 99 100 100 100

Before we leave the Shellsort algorithm altogether, it is probably a good idea to compare the two Shellsort algorithms we've discussed to see if one is more efficient than another. Here is a program for comparing the times of the two algorithms sorting a data set of 10,000 elements:

var nums = new CArray(10000);

nums.setData();

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

nums.shellsort();

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

var elapsed = stop - start;

print("Shellsort with hard-coded gap sequence: " + elapsed + " ms.");

nums.clear();

nums.setData();

start = new Date().getTime();

nums.shellsort1();

stop = new Date().getTime();

print("Shellsort with dynamic gap sequence: " + elapsed + " ms.");

The output is:

Shellsort with hard-coded gap sequence: 3 ms.

Shellsort with dynamic gap sequence: 3 ms.

Both algorithms sorted the data in the same amount of time. Here is the output with 100,000 data elements:

Shellsort with hard-coded gap sequence: 43 ms.

Shellsort with dynamic gap sequence: 43 ms.

Clearly, both of these algorithms perform the Shellsort algorithm with the same efficiency so you can use either of them with confidence.

## The Mergesort Algorithm

The Mergesort algorithm is so-named because it involves merging sorted sub-lists together to form a larger sorted list. In theory, this algorithm should be easy to implement. We need two sorted sub-arrays and a third array in which we merge the two sub-arrays by comparing data elements and inserting the smallest value from the two sub-arrays into the third array. In practice, the algorithm has some problems because if we are trying to sort a very large data set using Mergesort, the amount of space we need to store the merged sub-arrays can be quite large. Still, because space is not as big an issue these days as it used to be, with memory being so inexpensive, it is worth implementing the Mergesort algorithm to see how it compares in efficiency to other sorting algorithms.

### The Top-down Mergesort Algorithm

Our first implementation of the Mergesort algorithm will be the *top-down* *mergesort*. This algorithm is an example of a *divide-and-conquer* algorithm. The data set is recursively split into two halves until there are just a set of one-element arrays. These arrays are already sorted (since they contain just one element), so each array is merged into a final, sorted array containing all the data from the data set. The algorithm is called a top-down algorithm because the merging is performed at the bottom of the algorithm after all the recursive calls have been made. Figure x.2 illustrates how the top-down Mergesort algorithm works.