Shellsort

Although this sort doesn't have the fastest running time of all the sorting algorithms, the idea behind it is simple, yet it is fairly competitive with Quick and Merge sort for fairly decent sized data sets.

Here is the basic idea:

Rather than sorting all the elements at once, sort a small set of them, maybe every 5th element or so. (You can do this using an insertion sort.) Do this for 5 different sets of elements.

Then sort every 3rd element, etc.

Finally sort all the elements using insertion sort.

The rationale behind this sort is as follows:

A small insertion sort is quite efficient. A larger insertion sort can be efficient if the elements are already "close" to sorted order. By doing the smaller insertion sorts, the elements do get closer in order before the larger insertion sorts are done.

Here is an example of shell sort:

12 4 3 9 18 7 2 17 13 1 5 6

First let's do a 5-sort, meaning, let's sort every 5th element for the five groups of numbers. (The five groups are (12, 7, 5), (4, 2, 6), (3, 17), (9, 13), and (18, 1).) Place the numbers in their sorted locations, thus, the 5, 7 and 12 go in locations 0, 5 and 10.

5 2 3 9 1 7 4 17 13 18 12 6

Now, a 3-sort:

4 1 3 5 2 6 9 12 7 18 17 13

Finally a normal insertion sort will produce the sorted array:

1 2 3 4 5 6 7 9 12 13 17 18

Notice that by the time we do this last insertion sort, most elements don't have a long way to go before being inserted.

So now the question becomes, do we always do a 5, 3 and 1 sort? The answer is no. In general, we can see that shell sort will ALWAYS work as long as the last "pass" is a 1-sort.

The important question is, how do we space out the previous sorts. In particular, we'll call h_1, h_2, h_3, h_t, an increment sequence. For shellsort, first we will sort every h_t values using insertion sort, then every h_{t-1} values and so on...until we at last sort every h_1 values. (We must have h_1=1 as previously mentioned.)

What tends to work well is if each of the values in the increment sequence are in a geometric series. A good example would be 1, 2, 4, 8, 16, etc. Thus, if we were sorting 1000 values, our first sort may be a 256-sort. (Followed by a 128 sort, a 64 sort, etc.) Notice how quickly these initial "passes" will run. Generally, they will be O(n) time. As time goes on, they will be a bit slower, but not nearly as slow as the original insertion sort. In practice, it turns out that a geometric ratio of 2.2 produces the best results. (Roughly this would correspond to

the gap sequence 1, 2, 5, 11, etc.) The actual average case analysis of this sort is too difficult. (Our textbook states that experimental results indicate an average running time of $O(n^{1.25})$.)

A lower bound for sorts which swap adjacent elements only

An inversion in a list of numbers is a pair of numbers that are out of order relative to each other.

For example, in the list 7, 2, 9, 5, 4, the inversions are the following pairs: (7, 2), (7, 5), (7, 4), (9, 5), (9, 4), and (5,4). (Note that two elements can be inverted even if they are not adjacent to one another.)

Several sorts, such as insertion sort, only swap adjacent elements.

A key observation about any such sort is that in a single swap, only one inversion is eliminated, since all inversions with values outside the swap with the two values in the swap are maintained. (For example, if we swap 5 and 9 above, the inversion with 7 and 5 is still there.) A second observation is that a sorted list has no inversions. Putting these two facts together, we find that any such sorting algorithm in the category above, must have a minimum run-time proportional to the number of inversions in an unsorted list of numbers.

In a completely random list of n distinct numbers, the probability that any pair has an inversion is 0.5. There are a total of (n choose 2) pairs of values in a list of n distinct numbers. Thus, the average number of inversions in a random list of n distinct numbers is $\frac{1}{2}\binom{n}{2} = \frac{n(n-1)}{2} = \Omega(n^2)$**.**

It follows that the average case run-time of all of these algorithms is $\Omega(n^2)$**.**

A lower time bound for comparison based sorting

Given an input of n numbers to sort, they could be arranged in n! different orders. Clearly, for each of those n! cases, a sorting algorithm MUST "act" differently. In particular, if a sorting algorithm makes a certain number of comparisons, when running to each of these n! inputs, no two will give the same exact results for every comparison. (If they did, could you sort those two different data sets differently?)

Thus, we must make enough comparisons, in any comparison based sorting algorithm to distinguish between all n! inputs. A single comparison can distinguish between 2 separate inputs. (Because either an element is greater than the other, or less than the other. For the moment we are assuming distinct elements.) In general, k comparisons can distinguish between at most 2^k inputs. To see this, imagine making a chart like so:

Input	a[1]>a[2]	a[1]>a[3]	a[2]>a[3], ...	a[n-2]>a[n-1]
2,1,4,3	T	F	F	T
2,3,1,4	F	T	T	F
4,3,2,1	T	T	T	T

As you can see, if there are k columns in the chart, each with 2 possible answers, there are a total of 2^k possible distinct rows that could be on the chart. If two distinct inputs gave rise to the same EXACT row of answers, it would be impossible for our sorting algorithm to distinguish between the two inputs. Thus, if a sorting algorithm is to work, each input NEEDS to lead to a distinct row of answers.

This leads us to the equation:

$2^k > n!$

We need to find the smallest value of k which satisfies this equation above. Using logarithms we find this value to be $\log_2 n!$

Now, what is this value equal to approximately? It turns out that $n! > (n/e)^n$. So we find that

$2^k > n! > (n/e)^n$

$\log_2(2^k) > \log_2(n/e)^n$
$k > n(\log_2 n - \log_2 e)$
$k > n\log_2 n - n\log_2 e > n\log_2 n - 2n = \Omega(n\log_2 n).$

Thus, we have shown that it is necessary to make at least $\Omega(n\log_2 n)$ comparisons to sort n values in a purely comparison based sorting algorithm.

Bucket Sort

Although no comparison sort is faster than O(nlgn), if we do assume some information about the input that allows us to sort with extra information (not just comparisons), we can indeed improve our sorting time to O(n). (Clearly we can not do any better asymptotically because we must "look" at every number at least once to guarantee that the output list is sorted.)

In Bucket Sort, we will assume that the input values are randomly distributed in a range [0, N). (Our book says the values have to be integers, but the sort as well as the analysis will still work if we only assume that the values being sorted are real.)

Assume that we have n values to sort. We will create n different buckets to hold all the values. We can implement each bucket with a linked list. Each bucket will store values within a range of N/n.

Perhaps the easiest way to get a grasp of the algorithm is to go through an example:

Consider sorting a list of 10 numbers known to be in between 0 and 2, not including 2 itself. Thus, each bucket will store values in a range of 2/10 = .2 In particular, we have the following list:

Bucket	Range of Values	Bucket	Range of Values
0	[0, .2)	5	[1, 1.2)
1	[.2, .4)	6	[1.2, 1.4)
2	[.4, .6)	7	[1.4, 1.6)
3	[.6, .8)	8	[1.6, 1.8)
4	[.8, 1)	9	[1.8, 2)

Consider sorting the following list: 1.3, 0.7, 0.1, 1.8, 1.1, 1.0, 0.5, 1.7, 0.3, and 1.5. Here is a picture of what happens during the sort. Based upon this, imagine how one would write code to implement this sort. (The data structure to use would be an array of linked lists.)

Also, consider that it is not necessary for the input range to start at 0. As long as you have both a lower and upper bound for input, the bucket sort algorithm can be adapted.

Given that the range of inputs is [L, L+N), (where L stands for lowest possible value, and N stands for the range), and there are n values to be sorted, if we are given the value v, our first goal is to determine WHICH linked list to insert the value into. Once we do that, then we can simply call our linked list insert method to insert a value into a linked list in sorted order.

The correct array index is floor(n(v - L)/N), using integer division. To see this, consider that the value v-L is from the range [0, N). Now, we must map this to the correct array index from 0 to n-1. Each array index has a range of N/n values. Thus, what we want to do is divide our value v-L by the range of each bucket to give us the array index of the appropriate bucket. For example, using our previous list, if we were to sort 1.3, it would go to the bucket floor(10(1.3 - 0)/2) = 6 as desired.

Although I will skip the mathematics as to why this sort is O(n), I will give you the intuition as to why that is the case. Since we have as many buckets and values to sort, IF the values to sort are randomly distributed, then the length of each linked list will be constant with very high probability. Each insert in a constant sized linked list will take constant time, just the total amount of time for the sort will be linear in the number of items to sort.

Counting Sort

In counting sort, each of the values being sorted are in the range from 0 to m, inclusive. Here is the algorithm for sorting an array a[0],...,a[n-1]:

1) Create an auxiliary c, indexed from c[0] to c[m] and initialize each value in the array to 0.
2) Run through the input array a, tabulating the number of occurrences of each value 0 through m by adding 1 to the value stored in the appropriate index in c. (Thus, c is a frequency array.)
3) Run through the array c, a second time so that the value stored in each array slot represents the number of elements less than or equal to the index value in the original array a.
4) Now, run through the original input array a, and for each value in a, use the auxiliary array c to tell you the proper placement of that value in the sorted input, which will be stored in a new array b[0]..b[n-1].
5) Copy the sorted array from b to a.

Consider the following example:

index	0	1	2	3	4	5	6	7
A	3	6	4	1	3	4	1	4

First create the frequency array:

index	0	1	2	3	4	5	6
C	0	2	0	2	3	0	1

Now, to change C so that each array element stores the number of values less than or equal to the given index minus one, run the following loop:

```
C[0]--;
for i=1 to m
  C[i] = C[i] + C[i-1]
```

After this loop C loops like

index	0	1	2	3	4	5	6
C	-1	1	1	3	6	6	7

Now, to place A[7] in its sorted place in the new array B, simply look at what is stored in C[A[7]]. This is 6, which means that 4 ought to be placed in array index 6 of B. Also, we must decrement C[4] so that the next time we place another 4 in the array, it gets placed in a new location. Here are the changes to both B and C after placing A[8]:

index	0	1	2	3	4	5	6	7
B							4	

index	0	1	2	3	4	5	6
C	-1	1	1	3	5	6	7

Now, let's trace through the rest of this example...

Note that we go backwards through the array so that this sort is a stable sort. (What this means is that ties in the original input stay in the same relative order after being sorted. Namely, the last 4 in the input will be in array index 7 of the output, the second to last 4 in the input will be in array index 6 of the output, and the 4 in array index 3 of the input will be placed in index 5 of the output.)

One other thing to note, after arriving at the frequency array, we might say, "why didn't the text just loop through each index in C and place each corresponding number in the array A directly. (ie. Since C[1] = 1 originally, why not just place a 1 in A[0] and A[1] and move on...) The reason is that these numbers may have associated data with them, and in this latter approach, we wouldn't be placing that associated data with the numbers. The previous approach allows for this.

Radix Sort

The input to this sort must be non-negative integers all of a fixed length of digits. Let each number be k digits long. The sort works as follows:

1) Sort the values using a O(n) stable sort on the kth most significant digit.
2) Decrement k by 1
3) Repeat step 1. (Unless k=0, then you're done.)

Once again, this sort is much more easily understood with a demonstration. The running time of this sort should be O(nk), since we do k stable sorts that each run in O(n) time.

Depending on how many digits the numbers are, this sort can be more efficient than any O(nlgn) sort, depending on the range of values.

A stable sort is one where if two values being sorted, say v_i and v_j are equal, and v_i comes before v_j in the *unsorted* list, then v_i will STILL come before v_j in the *sorted* list.

This sort does seem quite counter intuitive. One question to ask: would it work the other way around (namely from most significant to least significant digit)? If not, could we adapt the sort to work the other way around? Why does this always work? Why does the intermediate sort need to be stable?

The key to it is as follows: After the mth iteration, the values are sorted in order with respect to the m least significant digits. Clearly, when m=k, this means the list is just plain sorted!

Here is an illustration of Radix sort:

unsorted			
----------	v	v	v
235	162	628	162
162	734	734	175
734	674	235	235
175	235	237	237
237	175	162	628
674	237	674	674
628	628	175	734

The second column is sorted by the units digit, the third by the tens digit and the last column by the hundreds digit. Notice how each sort is stable, (when there are ties, the order of the elements is NOT switched. Only in this case is correctness guaranteed.)