Shell Sort

Introduction

Most of the sorting algorithms seen so far such as insertion sort and selection sort have a run time complexity of O(N^2). We also know that no sorting algorithm can have time complexity less than O(N), since we need to scan the list at least once. For example, insertion sort best case complexity is O(N). Can we get a better performance and get run time complexity between O(N^2) and O(N)? The answer is Yes and there are many algorithms with a complexity in this range, with varying tradeoffs. In this note, we describe one approach: Shell Sort. Shell Sort has evolved as a trial and error algorithm. Analytical results are not available but empirically it has been found that the Shell Sort improves the efficiency and decreases the run time complexity to O($N^{1.25)}$. Donald Shell discovered the Shell sort and thence it is known as Shell Sort.

Algorithm

The algorithms for shell sort can be defined in two steps

Step 1: divide the original list into smaller lists.
Step 2: sort individual sub lists using any known sorting algorithm (like bubble sort, insertion sort, selection sort, etc).

Many questions arise ☺: how should I divide the list? Which sorting algorithm to use? How many times I will have to execute steps 1 and 2? And the most puzzling question if I am anyway using bubble, insertion or selection sort then how I can achieve improvement in efficiency? We will discuss each of these questions one by one.

For dividing the original list into smaller lists, we choose a value *K*, which is known as *increment.* Based on the value of *K*, we split the list into *K* sub lists. For example, if our original list is x[0], x[1], x[2], x[3], x[4]….x[99] and we choose 5 as the value for increment, *K* then we get the following sub lists.

first_list = x[0], x[5], x[10], x[15]…….x[95]
second_list =x[1], x[6], x[11], x[16]…….x[96]
third_list =x[2], x[7], x[12], x[17]…….x[97]
forth_list =x[3], x[8], x[13], x[18]…….x[98]
fifth_list =x[4], x[9], x[14], x[19] …….x[99]

So the i^{th} sub list will contain every K^{th} element of the original list starting from index i-1.

According to the algorithm mentioned above, for each iteration, the list is divided and then sorted. If we use the same value of *K*, we will get the same sub lists and every time we will sort the same sub lists, which will not result in the ordered final list. Note that sorting the five sub lists independently do not ensure that the full list is sorted! So we need to change the value of *K* (increase or decrease?) for every iteration. To know whether the array is sorted, we need to scan the full list.

We also know that number of sub lists we get are equal to the value of *K*. So if we decide to reduce the value of *K* after every iteration we will reduce the number of sub lists also in every iteration. Eventually, when *K* will be set to 1, we will have only one sub list. Hence we know the termination condition for our algorithm is *K* = 1. Since for every iteration we are decreasing the value of the increment (*K*) the algorithm is also known as "*diminishing increment sort*".

Any sorting algorithm or a combination of algorithms can be used for sorting the sub lists, e.g. some shell sort implementation use bubble sort for the last iteration and insertion sort for other iterations. We use insertion sort in this tutorial. How does shell sort give better performance if the sorting is ultimately done by algorithms like insertion sort and bubble sort? The rationale is as follows.

If the list is either small or almost sorted then insertion sort is efficient since less number of elements will be shifting. In Shell Sort as we have already seen, the original list is divided into smaller lists based on the value of increment and sorted. Initially value of K is fairly large giving a large number of small sub lists. We know that simple sorting algorithms like insertion sort are effective for small lists, since the data movements are over short distances. As K reduces in value, the length of the sub lists increases. But since the earlier sub lists have been sorted, we except the full list to look more and more sorted. Again note that algorithms such as insertion sort are fairly efficient when they work with nearly sorted lists. Thus the inefficiency arising out of working with larger lists is partly compensated by lists being increasingly sorted. This is the intuitive explanation for the performance of shell sort.

How to choose the value of increment K?

Till date there is no convincing answer to what should be the optimal sequence for the increment, K. But it has been empirically found that larger number of increments gives more efficient results. Also it is advisable not to use empirical sequences like 1,2,4,8… or 1,3,6,9… if we do so, for different iteration, we may get almost the same elements in the sub lists to compare and sort which will not result in better performance. For example, consider a list X= 12 6 2 5 8 10 1 15 31 23 and increment sequences 6 and 3.

When the increment is 6, we will get the following sub lists

(12,1)
(6,15)
(2,31)
(5,23)
(8)
(10)

After first iteration
X=1 6 2 5 8 10 12 15 31 23

When the increment reduces to 3,we will get the following lists

(1, 5, 12, 23)
(6, 8, 15)
(2, 10, 31)

Now if we analyse the first sub lists where increment was 6 and then second sub lists where increment was 3, we can see that in the second set of lists, we are comparing and sorting 1, 12 again which we had already sorted before in the previous sub list. So, we should choose the value of *K* in such a way that in every iterations we get almost different elements in the sub lists to compare and sort. There are proven and recommended ways for generating the increment sequences. Some of them have been discussed below.

Donald Shell has suggested

$H_t = N/2$

$H_k = h_{k+1} /2$

Which give the increment series as: N/2, N/4, …….1

Hibbard suggested the sequence of increment as 1, 3, 7, …….2^K-1.

Another way for calculating increment K suggested by Knuth and we have used this method for generating the increments in this tutorial.

hi = 1

hi = hi* 3 +1 and stops at $h_{t,}$ When h_{t+2} >= N.

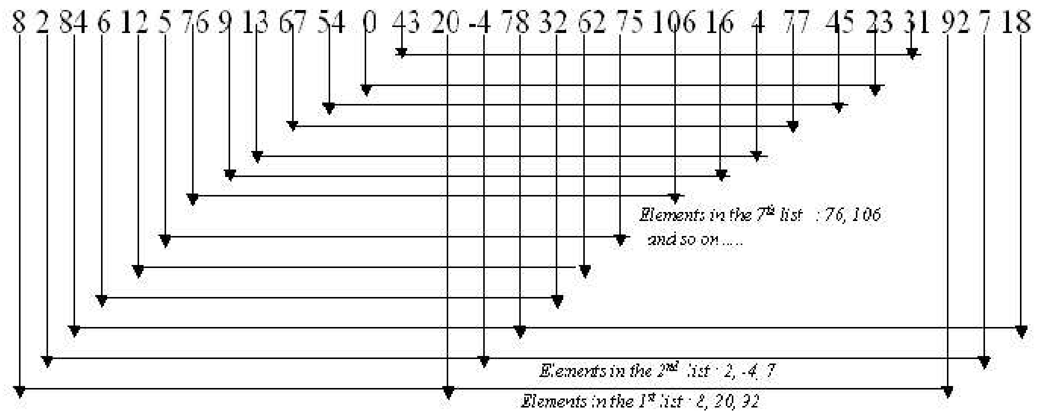
Which gives the increment series as 1, 4, 13….. $h_{t.}$

Other than these, there are many series which have been empirically found and perform well.

Example

We will take an example to illustrate Shell sort. Let the list be
8 2 84 6 12 5 76 9 13 67 54 0 43 20 -4 78 32 62 75 106 16 4 77 45 23 31 92 7 18
and the increment values using Knuth formula we get are 13, 4, 1.

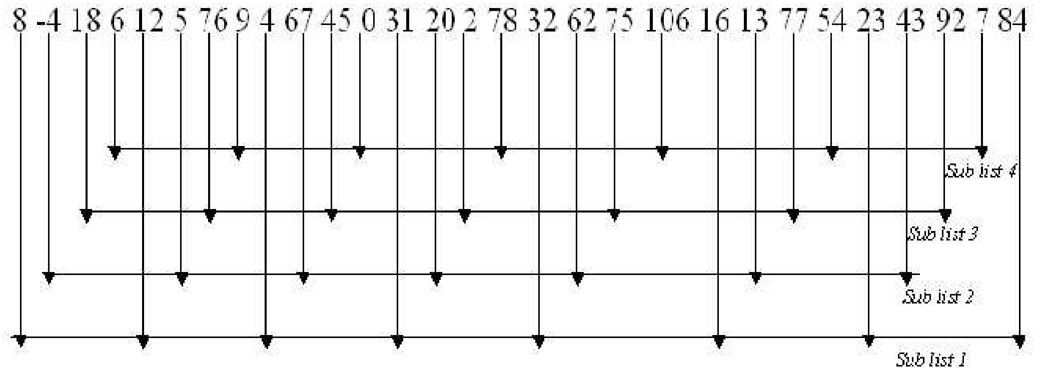
When increment is 13, we get the following 13 sub lists and the elements are divided among the sub lists in the following ways i.e. every K^{th} element of the list starting with the index i will be in the sub list i.


8 2 84 6 12 5 76 9 13 67 54 0 43 20 -4 78 32 62 75 106 16 4 77 45 23 31 92 7 18
Elements in the 7th list : 76, 106
and so on…..
Elements in the 2nd list : 2, -4, 7
Elements in the 1st list : 8, 20, 92

After sorting the sub lists, the resulting list we get is as follows.

8 -4 18 6 12 5 76 9 4 67 45 0 31 20 2 78 32 62 75 106 16 13 77 54 23 43 92 7 84

We now, reduce the increment value to 4 and get the following 4 sub lists.



After sorting these sub lists, the resulting list is as follows:

4 -4 2 0 8 5 18 6 12 13 45 7 16 20 75 9 23 43 76 54 31 62 77 78 32 67 92 106 84

We further reduce increment value to 1, which is our last increment value. Now there is only one list, which after sorting, we get:

-4 0 2 4 5 6 7 8 9 12 13 16 18 20 23 31 32 43 45 54 62 67 75 76 77 78 84 92 106

Now try to understand and analyse what is happening? How the elements are moved? We can see, after the first iteration when the increment was 13, the element –4 of the list, which was at 15th position in the original list has reached to the 2nd position. The element 18 was at 29th position and has reached the 3rd position. Using insertion sort on the whole list, we know that if –4 has to reach from the 15th position to the 2nd position, the required number of movements of the elements in the list will be very high. Shell Sort performs better than insertion sort by reducing the number of movements of elements in the lists. This is achieved by large strides the element take in the beginning cycle.

Java Implementation for Shell Sort

```
/*
Shell Sort Algorithm.

Input Instructions:
            reads the input numbers till -1 is read.

Output Instructions:
            Gives the increment and the sorted lists after each increments.

Functions:
            readElements();
            generateIncrementSeq();
            solve();
            insertionSort();
```

```
        print();
*/


import ncst.pgdst.*;

class ShellSort {
    SimpleInput sin;
    SimpleOutput sout;
    int N;     // the number of elements in array
    int a[];   // the array of numbers to be sorted
    int inc[]; // the array of increments
    int noOfInc; // total number of increments

/* read all the input numbers till -1 is read */
void readElements() throws IOException{
    a=new int[1000];
    N=0;
    sin=new SimpleInput();
    sout=new SimpleOutput();
    sin.skipWhite();
    int val=sin.readInt();

    while(val != -1)  {
        a[N]=val;
        N++;
        sin.skipWhite();
        val=sin.readInt();
    }//while
    print();
}

/* Generate increment sequences. Uising Knuth sequences */
void generateIncrementSeq() {

    inc=new int[1000];
    noOfInc=0; //to keep track of the number of increments generate.
    for(int h=1;h<N;) {
        inc[noOfInc]=h;
        h=3*h + 1;
        noOfInc++;
    }//for
}


/*
1. Reads the Input, by calling the method reasElemnets().
2. Generates the increment sequence,by calling the method generateIncrementSeq()
3. Then for each increment performs insertion sort with each sublist
*/
```

```
void solve() throws IOException{

        readElements();
        generateIncrementSeq();

        // start with the largest increment and proceed till 1.
        for(int k=noOfInc-1;k>=0;k--) {
             int increment=inc[k];

             sout.writelnString("increment "+increment);
             // here the list is divided into 'increment' subarrays
             for (int sublistid=0; sublistid < increment; sublistid++)
                 //perform insertion sort with each sublist
                 //note that you can use any sorting algorithm here
                 insertionSort(increment, sublistid);
            print();
        }//for k
}//solve

public void insertionSort(int increment, int sublistid) {
        // outer loop for a given sublist, starting at index sublistid
        for(int i=increment+sublistid; i<N; i += increment) {
                int value=a[i];
                int j;
                // inner loop in which each element greater than the current
               // is pushed behind by 'increment' positions
              // Note that only elements at 'increment' distance apart from each other are
            compared and moved
                for(j=i-increment;j>=0 && value<a[j]; j-=increment) {
                        a[j+increment]=a[j];
                }//for j
                // swap only if movement is made
                if(j-increment != i) {
                        a[j+increment]=value;
                }
        }//for i
}//insertionSort

/* prints the list */
void print() {
        for(int i=0;i<N;i++) {
                sout.writeInt(a[i]);
                sout.writeChar(' ');
        }
        sout.writeln();
}//print

public static void main(String args[]) throws IOException {
        ShellSort ss=new ShellSort();
        ss.solve();
}//main

}//class ShellSort
```

Complexity Analysis

It has been empirically found that most of the shell sort versions perform better than simple sorting algorithms such as insertion sort, bubble sort, selection sort. Analytical results are not available.

The average case complexity of shell sort empirically found is O($N^{1.25}$). The worst case runtime depending on the sequence of increments we chose may vary from O($N^{1.25}$) to O(N^2).