PROJECT REPORT

DESCRIPTION OF SHELL SORT AND EMPIRICAL EVALUATION OF PERFORMANCE

PRITAM DEY
AYAN PAUL
Guided By - Prof. Deepayan Sarkar
Indian Statistical Institute, Delhi

Contents

1 What Is Shellsort? 3

2 Algorithm 3

3 Visualisation 4

4 Useful Definitions Related To Shellsort 5

5 Usefulness of Shellsort 6

6 Shellsort and Frobenius Coin Problem 6
6.1 Frobenius Problem 6
6.2 Mathematical Statement of The Problem 6
6.3 Existence of Frobenius Number 6
6.4 Frobenius Number for Small Set of Integers 7
6.5 Relation with Shell sort 7

7 Time Complexity 8

8 Gap Sequences 9

9 Ciura's Sequential Analysis 13

10 Comparison of different methods 15
10.1 Methods used to calculate different measurements 16
10.2 Number of comparisons 16
10.3 Number of swaps/assignments 19
10.4 Time 21

11 Limitations 23

12 Conclusions 23

1 What Is Shellsort?

Shellsort is mainly a variation of **Insertion sort**. In Insertion sort, we move elements only one position ahead. When an element has to be moved far ahead, many movements are involved.
The idea of Shellsort is to allow exchange of far items. In Shellsort, we make the array h-sorted for a large value of h. We keep reducing the value of h until it becomes 1. An array is said to be h-sorted if all sublists of every h^{th} element is sorted.
Donald Shell published the first version of this sort in 1959. [1]

2 Algorithm

```
1  Function ShellSort (Array, GapSequence):
       // Start with the largest gap and work down to a gap of 1
2      foreach gap ∈ GapSequence do
           // Do a gapped insertion sort for this gap size.
           // The first gap elements Array[0...gap-1] are already
              in gapped order
           // keep adding one more element until the entire array
              is gap-sorted
3          for i ← gap to (n − 1) do
               // save Array[i] in key and make a hole at position
                  i
4              key ← Array[i]
5              for j ← (i − gap) to 0 by −gap do
                   // shift earlier gap-sorted elements up until
                      the correct location for Array[i] is found
6                  if Array[j] < key then
7                      break
8                  else
9                      Array[j + gap] ← Array[j]
10                 end
11             end
               // put key (the original Array[i]) in its correct
                  location
12             Array[j + gap] ← key
13         end
14     end
15 end
```

3 Visualisation

Figure 1: Sorting sub-arrays of gap 4

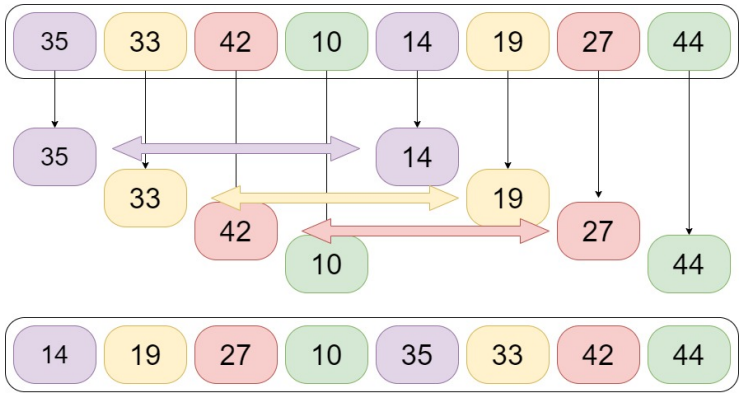


Figure 2: Sorting sub-arrays of gap 2

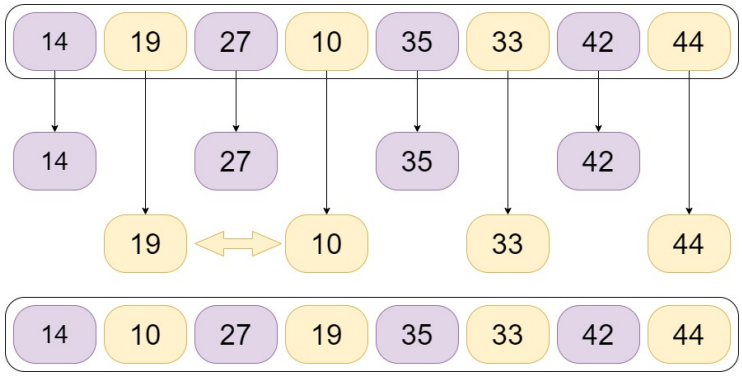
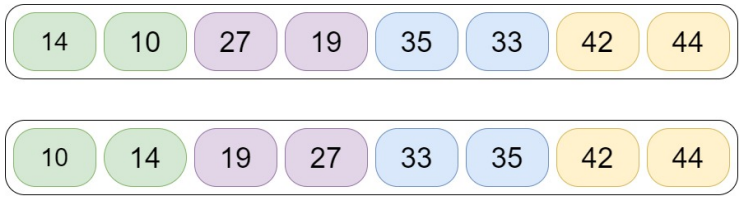


Figure 3: Sorting sub-arrays of gap 1 (Insertion Sort)



4 Useful Definitions Related To Shellsort

Pass

Each trial of the Shellsort algorithm where every possible lists(sub-arrays) of gap h between any two elements of the list is called a **Pass**.

E.g. : Referring to the example shown in the **Visualisation** section, sorting sub-arrays of gap 4, gap 2 and gap 1 are the first, second and third passes of the Shellsort algorithm respectively.

h sort

At every pass, the Shellsort algorithm sorts every sub array or list of elements having gap h between any two of them in the original array. So after each pass, the algorithm yields some h interleaved lists, each individually sorted. This process is called **h-sorting**. Beginning with large values of h, this rearrangement allows elements to move long distances in the original list, reducing large amounts of disorder quickly and leaving less work for smaller h-sort steps to do. If the list is then k-sorted for some smaller integer k, then the list remains h-sorted. Following this idea for a decreasing sequence of h values ending in 1 is guaranteed to leave a sorted list in the end.

E.g. : Referring to the example shown in the **Visualisation** section, in the first pass h is 4, in the second pass h equals to 2 and in the third pass h equals to 1.

Gap Sequence

It is a proposed sequence of integers which determines which h-sortings will be done by the algorithm to sort the whole array. These are mostly random or experimentally generated integers. Use of a good gap sequence reduces the time complexity of the algorithm.

E.g. : Referring to the example shown in the **Visualisation** section, the gap sequence used is {4, 2, 1}.

5 Usefulness of Shellsort

Applications of **Shellsort** includes:

- **Shellsort** performs more operations and has higher cache miss ratio than **Quicksort**.
- However, since it can be implemented using little code and does not use the call stack, some implementations of the **Quicksort** function in the C standard library targeted at embedded systems use it instead of **Quicksort** . **Shellsort** is, for example, used in the uClibc library. For similar reasons, an implementation of **Shellsort** is present in the Linux kernel.
- **Shellsort** can also serve as a sub-algorithm of introspective sort, to sort short subarrays and to prevent a slowdown when the recursion depth exceeds a given limit. This principle is employed, for instance, in the bzip2 compressor.

6 Shellsort and Frobenius Coin Problem

6.1 Frobenius Problem

There is a nice relation between Shell Sort, its gap sequences and a renowned problem in Number Theory, **The Frobenius Coin Problem**.
The problem asks for the largest monetary amount that cannot be obtained using only coins of specified denominations. For example, the largest amount that cannot be obtained using only coins of 3 and 5 units is 7 units. The solution to this problem for a given set of coin denominations is called the **Frobenius Number** of the set.
The Frobenius number exists as long as the set of coin denominations are coprimes.

6.2 Mathematical Statement of The Problem

In mathematical terms the problem can be stated as:
Given positive integers $a_1, a_2, \ldots, a_n$ such that $gcd(a_1, a_2, ..., a_n) = 1$, find the largest integer that cannot be expressed as an integer conical combination of these numbers, i.e. as a sum $k_1a_1 + k_2a_2 + \cdots + k_na_n$, where $k_1, k_2, \ldots, k_n$ are non-negative integers.
This largest integer is called the **Frobenius Number** of the set a_1,a_2,...,a_n, and is usually denoted by g(a_1,a_2,...,a_n). [2]

6.3 Existence of Frobenius Number

The requirement that the greatest common divisor (GCD) equal 1 is necessary in order for the **Frobenius number** to exist. If the GCD were not 1, then starting at some number m the only sums possible are multiples of the GCD;

every number past m that is not divisible by the GCD cannot be represented by any linear combination of numbers from the set.
For example, if you had two types of coins valued at 6 cents and 14 cents, the GCD would equal 2, and there would be no way to combine any number of such coins to produce a sum which was an odd number. Also even numbers 2, 4, 8, 10, 16 and 22 (less than m=24) could not be formed, either.
On the other hand, whenever the GCD equals 1, the set of integers that cannot be expressed as a conical combination of $(a_1, a_2, \ldots, a_n)$ is bounded and therefore the **Frobenius number** exists.

6.4 Frobenius Number for Small Set of Integers

A closed-form solution exists for **The Coin Problem** only where n = 1 or 2. No closed-form solution is known for n > 2.

- **n=1**

  If n = 1, then $a_1 = 1$ so that all natural numbers can be formed. Hence no **Frobenius number** in one variable exists.

- **n=2**

  If n = 2, the **Frobenius number** can be found from the formula $g(a_1, a_2) = a_1a_2 - a_1 - a_2$. This formula was discovered by **James Joseph Sylvester** in 1882. **Sylvester** also demonstrated for this case that there are a total of $N(a_1, a_2) = \frac{(a_1-1)(a_2-1)}{2}$ non-representable (non-negative) integers.

- **n=3**

  Formulae and fast algorithms are known for three numbers though the calculations can be very tedious if done by hand.

  Simpler lower and upper bounds for **Frobenius numbers** for n = 3 have been also determined. The asymptotic lower bound due to **Davison** $f(a_1, a_2, a_3) \equiv g(a_1, a_2, a_3) + a_1 + a_2 + a_3 \geqslant \sqrt{3a_1a_2a_3}$

6.5 Relation with Shell sort

The worst case complexity of **Shell sort** has an upper bound which can be given in terms of the **Frobenius Number** of a given sequence of positive integers. [3]

7 Time Complexity

Time complexity of Shell sort is directly related to the gap sequence used in the algorithm. Shell originally proposed the following gap sequence. (1959)

$$\left[\frac{N}{2}\right], \left[\frac{N}{4}\right], \ldots, 1$$

Now let us look at some situation where this gap sequence may perform badly. Consider the gap sequence $\{25, 35, 14, 34, 26, 30, 10, 44\}$. Then just before the last pass, the array will undergo through these following changes.

Figure 4: Sorting sub-arrays of gap 4

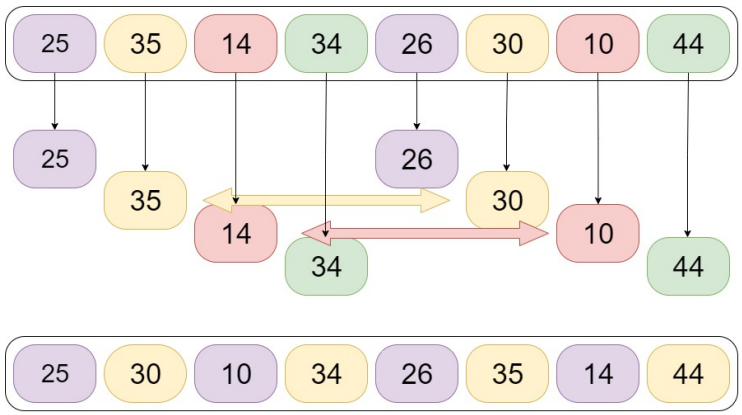
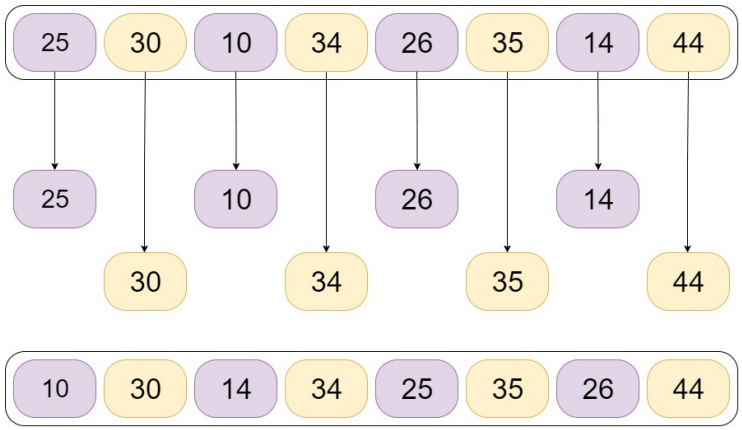


Figure 5: Sorting sub-arrays of gap 4



Now the problem becomes very clear as all the large numbers are placed in the even positions and all the smaller numbers are placed in odd positions. In

fact, Shell's proposed gap sequence compares the elements at the even position to the elements of the odd position only at the very last pass. Hence, we can construct such an array where all the large elements end up being at the even positions just before the last pass. Hence it suffers the disadvantages of insertion sort with large arrays.

Thus the question of deciding which gap sequence to use is hence very important. A carefully selected gap-sequence can improve the time complexity of Shell Sort significantly. Following are the properties of some good gap sequences.

- Any gap sequence that includes 1 yields a correct sort.
- Given the length of the array, the gap sequence should not be too large or too small.
- The gap sequence should have pairwise co-prime members.

  Sedgewick recommends to use gaps that have low greatest common divisors or are pairwise co-prime. It is quite intuitive since if we have a gap sequence like $\{1, 3, 5, 8\}$. Then after 8-pass and 5-pass, 3-sorting for the array will always be efficient. Moreover, we know that after 3-sorting and 5-sorting, an array gets 8-sorted $(1 \cdot 3 + 1 \cdot 5)$ automatically. However, the elements at gaps 2, 4 or 7 will never be compared.
  So it is better to construct a gap sequence with pairwise co-prime integers.
- Gonnet and Baeza-Yates observed that Shell sort makes the fewest comparisons on average when the ratios of successive gaps are roughly equal to 2.2. Whereas Tokuda suggested gap-sequence with successive ratio 2.25 is more efficient.

8 Gap Sequences

The question of deciding which gap sequence to use is difficult. Every gap sequence that contains 1 yields a correct sort (as this makes the final pass an ordinary insertion sort); however, the properties of thus obtained versions of Shell sort may be very different. Too few gaps slow down the passes and too many gaps produce an overhead.
Following are the most proposed gap sequences published so far. Some of them have decreasing elements that depend on the size of the sorted array (N). Others are increasing infinite sequences, whose elements less than N should be used in reverse order.

Shell's Sequence

- **Year of Publication**: 1959
- **General Term**: $\lfloor \frac{N}{2^k} \rfloor$
- **Concrete Gaps**: $\lfloor \frac{N}{2} \rfloor, \lfloor \frac{N}{4} \rfloor, ..., 1$
- **Worst Case Time Complexity**: $\Theta(N^2)$

Frank and Lazarus's Sequence

- **Year of Publication**: 1960
- **General Term**: $2\lfloor \frac{N}{2^{k+1}} \rfloor + 1$
- **Concrete Gaps**: $2\lfloor \frac{N}{4} \rfloor + 1, ..., 3, 1$
- **Worst Case Time Complexity**: $\Theta(N^{\frac{3}{2}})$

Hibbard's Sequence

- **Year of Publication**: 1963
- **General Term**: $2^k - 1$
- **Concrete Gaps**: $1, 3, 7, 15, 31, 63, ...$
- **Worst Case Time Complexity**: $\Theta(N^{\frac{3}{2}})$

Papernov and Stasevich's Sequence

- **Year of Publication**: 1965
- **General Term**: $2^k + 1,\ prefixed\ with\ 1$
- **Concrete Gaps**: $1, 3, 5, 9, 17, 33, 65, ...$
- **Worst Case Time Complexity**: $\Theta(N^{\frac{3}{2}})$

Pratt's Sequence

- **Year of Publication**: 1971
- **General Term**: $Successive\ numbers\ of\ the\ form\ 2^p3^q\ (3-smooth\ Numbers)$
- **Concrete Gaps**: $1, 2, 3, 4, 6, 8, 9, 12, ...$
- **Worst Case Time Complexity**: $\Theta(N \log_2 N)$

Knuth's Sequence

- **Year of Publication**: 1973, based on **Pratt**'s sequence.
- **General Term**: $\frac{3^k-1}{2}, not\ greater\ than\ \lceil \frac{N}{3} \rceil$
- **Concrete Gaps**: $1, 4, 13, 40, 121, ...$
- **Worst Case Time Complexity**: $\Theta(N^{\frac{3}{2}})$

Incerpi and Sedgewick's Sequence

- **Year of Publication**: 1985
- **General Term**:
  $\prod_I a_q,\ where\ a_q = min\{n \in \mathbb{N} : n \geqslant (5/2)^{q+1}\ \forall p : 0 \leqslant p < q \implies gcd(a_p, n) = 1\}$
  $I = \{0 \leqslant q < r |\ q \neq \frac{(r^2+r)}{2} - k\}$
  $r = \lfloor \sqrt{2k + \sqrt{2k}} \rfloor$
- **Concrete Gaps**: $1, 3, 7, 21, 48, 112, ...$
- **Worst Case Time Complexity**: $\Theta(N^{1+\sqrt{\frac{8\log(\frac{5}{2})}{\log N}}})$

Sedgewick's First Sequence

- **Year of Publication**: 1982

- **General Term**: $4^k + 3.2^{k-1} + 1,\ prefixed\ with\ 1$
- **Concrete Gaps**: $1, 8, 23, 77, 281, ...$
- **Worst Case Time Complexity**: $\Theta(N^{\frac{4}{3}})$

Sedgewick's Second Sequence

- **Year of Publication**: 1986
- **General Term**:
  $9(2^k - 2^{k/2}) + 1;\ k\ even$
  $8.2^k - 6.2^{(k+1)/2} + 1;\ k\ odd$
- **Concrete Gaps**: $1, 5, 19, 41, 109, ...$
- **Worst Case Time Complexity**: $\Theta(N^{\frac{4}{3}})$

Gonnet and Baeza-Yates's Sequence

- **Year of Publication**: 1991
- **General Term**:
  $h_k = max\{\lfloor\frac{5h_{k-1}}{11}\rfloor, 1\},\ h_0 = N$
- **Concrete Gaps**: $\lfloor\frac{5N}{11}\rfloor, \lfloor\frac{5}{11}\lfloor\frac{5N}{11}\rfloor\rfloor, ..., 1$
- **Worst Case Time Complexity**: $Unknown$

Tokuda's Sequence

- **Year of Publication**: 1992
- **General Term**: $\lceil\frac{1}{5}(9.(\frac{9}{4})^{k-1} - 4)\rceil$
- **Concrete Gaps**: $1, 4, 9, 20, 46, 103, ...$
- **Worst Case Time Complexity**: $Unknown$

Ciura's Sequence

- **Year of Publication**: 2001
- **General Term**: Unknown(Experimentally Derived)
- **Concrete Gaps**: $1, 4, 10, 23, 57, 132, 301, 701$
- **Worst Case Time Complexity**: *Unknown*

9 Ciura's Sequential Analysis

To find a decent gap-sequence for Shell Sort, Marcin Ciura from Poland performed sequential analysis with large arrays in the year 2001. Ciura showed that the number of comparisons made by Shell sort algorithm for a particular gap sequence follows approximately a normal distribution.

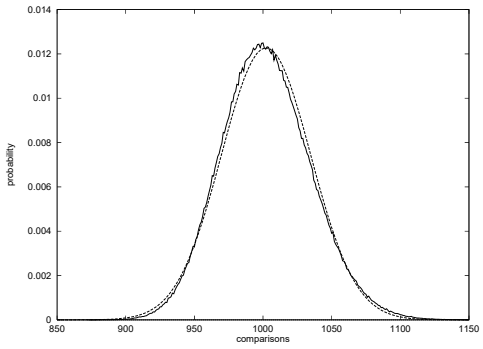


Fig. 2. Distribution of the number of comparisons in Shellsort using the sequence $(1, 4, 9, 24, 85)$ for sorting 128 elements (*solid line*), and the normal distribution with the same mean and standard devistion (*dashed line*)

- In particular for sorting 128 elements, he observed that average number of comparisons (EC) made by a good gap sequence can be less than 1005, with maximum standard deviation 40.

- So he constructed a sequential test, that accepts a gap sequence that has $EC < 1005$ and rejects a sequence with $EC > 1015$. If EC lies between 1005 and 1015, then he would perform the test one more time.
- Setting $\alpha = 0.01$ (accidental rejection of a good sequence) and $\beta = 0.01$ (accidental acceptance of a bad sequence) he continued the test procedure for gap sequences of length $2, 3, 4, \ldots$
- He also stated that the standard deviations for the best sequences are always similar and to compensate the fact that actual distribution of number of comparisons is slightly skewed than the theoretical distribution, he chose a standard deviation larger than the actual standard distribution.

Ciura claimed[4] that comparisons rather than moves should be considered the dominant operation in Shellsort. And searched for a sequence which has fewer comparisons than any other sequence for arrays of size up to 8000. Here are the results of his experiment.

Table 4. The best 8-increment beginnings of 10-pass sequences for sorting 8000 elements

Increments	Ratio passed
1 4 10 23 57 132 301 758	0.6798
1 4 10 23 57 132 301 701	0.6756
1 4 10 21 56 125 288 717	0.6607
1 4 10 23 57 132 301 721	0.6573
1 4 10 23 57 132 301 710	0.6553
1 4 9 24 58 129 311 739	0.6470
1 4 10 23 57 132 313 726	0.6401
1 4 10 21 56 125 288 661	0.6335
1 4 10 23 57 122 288 697	0.6335

Here Ratio passed column represents the percentage of larger sequences starting with the given sequence has passed his sequential test

Finally based on his experience he conjectured that the sequence beginning with $\{1, 4, 10, 23, 57, 132, 301, 701\}$ shall turn up optimal.

10 Comparison of different methods

Efficiency of a sorting algorithm depends on many different factors. But it mostly depends upon number of comparisons and assignments made to sort an array. We have mainly considered these two measures for various sorting methods in this project. We compared efficiency of different gap sequences and then compared Shell sort with Quick sort using the best gap sequences for Shell sort. Finally we compared time taken by each method to sort arrays of different lengths. We have used three different plots to compare the sorting methods. We have plotted the following variables against length of array in each plot.

- $$\frac{Average\ number\ of\ comparisons}{log_2 N!}$$
- $$\frac{Average\ number\ of\ element\ swaps\ or\ value\ assignment}{log_2 N!}$$
- $$\frac{Average\ time\ taken\ by\ the\ algorithm}{NlogN}$$

In this project we shall only compare 4 best known gap sequences till now proposed by different individuals. These are the gap sequences proposed by Ciura, Tokuda, Gonnet and Baeza-Yates, and Sedgewick. Through out the next few plots we shall denote these gap sequences in the following way

- Ciura(2001): CI01
- Tokuda(1992): TO92
- Gonnet and Baeza-Yates(1991): GB91
- Sedgewick(1986): SE86

  also we shall denote the Quick sort algorithms with different partitioning schemes in the following way

- Lomuto Partitioning Scheme: QuickSortL
- Hoare's Partitioning Scheme: QuickSortH

Note that the sequential analysis done by Ciura is for arrays with size as much as 8000. But to compare its efficiency for even larger sizes we extended the sequence proposed by Ciura using Tokuda's proposed ratio i.e. the sequence $\{1, 4, 10, 23, 57, 132, 301, 701\}$ is extended using the formula $h_k = [2.25h_{k-1}]$. Where

10.1 Methods used to calculate different measurements

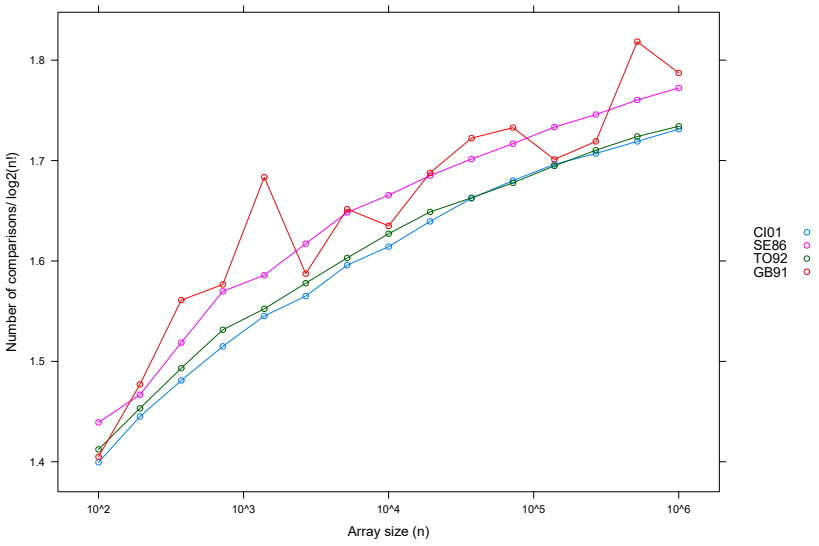
Number of comparisons, assignments/swaps and running time may vary widely depending upon the randomness of the array. So we used the following methods to obtain different measurements-

- To compare the number of comparisons (assignments/swaps) of different sorting algorithms, we produced 20 different random arrays for each given length and observed average number of comparisons (assignments/swaps) made by different algorithms. Each algorithm was given same set of random arrays to ensure fair comparison.
- Running time of a sorting algorithm depends on various subtle factors like allocation of memory, processor speed, amount of available memory etc. So, for each given length we generated 20 different random arrays, and each of these arrays were sorted 5 times by each algorithm. Finally, the grand means (in micro seconds) were observed for different algorithms. Again, to ensure fair comparison same set of inputs were given to every algorithm.

10.2 Number of comparisons

We start by comparing number of comparisons of different gap sequences

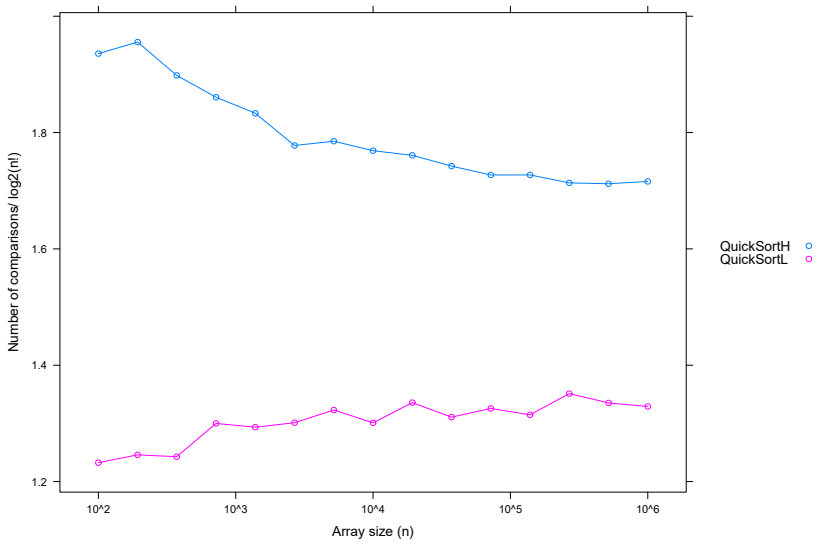
Figure 6: Number of comparisons for different sequences



Observations:

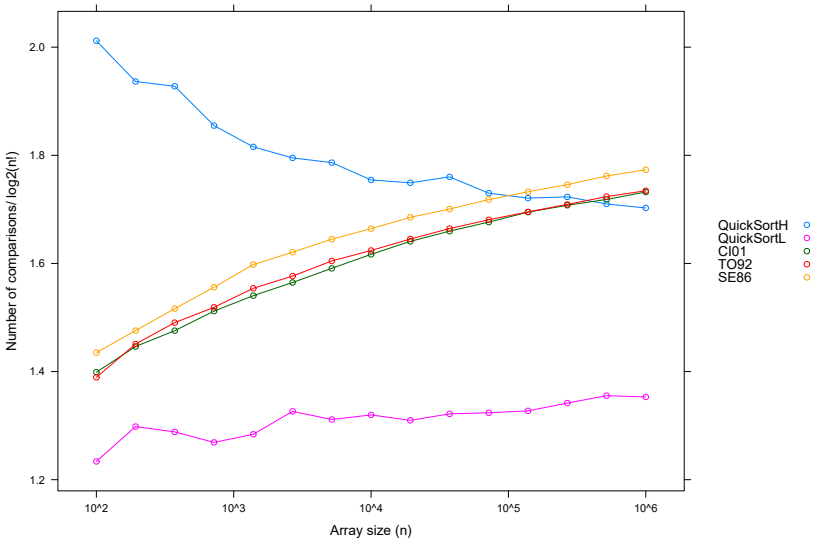
- CI01 performs consistently better than other gap sequence in terms of comparison. However when array size exceeds 10^4 it behaves similarly to the TO92. This is an expected behaviour since Ciura originally experimented up to array size 8000 and we extended the sequence using Tokuda's proposed ratio 2.25.
- Tokuda's proposed ratio(2.25) for successive gaps seems to be more efficient than the one proposed by Gonnet and Baeza-Yates(2.2). Also TO92 has a closed form expression.
- Out of the 12 best known gap sequences, only 3 of them depends on the exact size of the array. One of them being Shell's proposed sequence. Although GB91 performs much better than the other two, number of comparisons seems to vary significantly with increase of array sizes. GB91 performs very close to TO92 for some array sizes, whereas for other sizes it performs very poorly. Hence it is very hard to conclude anything for GB91. It seems that the gap sequences that depends on the exact size of the array are not good in general for different lengths of array. Hence we will avoid GB91 for sub sequence analysis of comparisons.
- SE86 makes more comparison in general than TO92 and CI01, but the order (curve) is very similar to the other sequences.

Figure 7: Number of comparisons for different partioning schemes



Observations: We can see that Lomuto partitioning scheme is significantly better than that of Hoare's.

Figure 8: ShellSort vs QuickSort — Number of comparisons



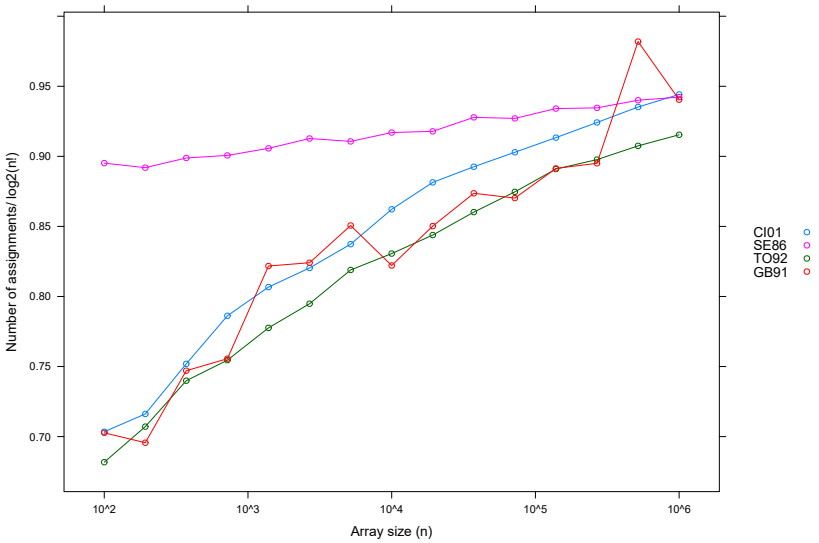
Observations:

- We can clearly see that Lomuto partitioning scheme is better than any other methods in terms of number of comparisons.
- Shell sort with best gap sequences perform less comparisons than Hoare's partitioning scheme. However, for array sizes greater than 10^5, Hoare's partitioning scheme has less comparisons.
- Considering the number of comparisons made by two different partitioning schemes of Quick sort, TO92 and CI01 is almost similar.

10.3 Number of swaps/assignments

We have implemented Shell sort in a way such that it does not involve any swaps. But still assignments of values make a great impact over the efficiency of any sorting algorithm. We can say that one swap is equal to three assignments and will compare the number of swaps/assignments for different methods.

Figure 9: Number of assignments for different gap sequences

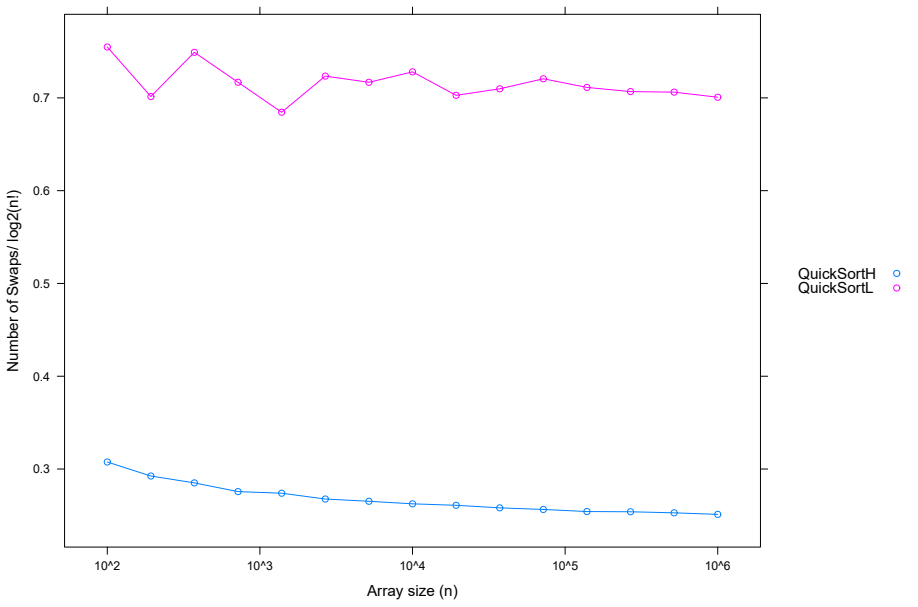


Observations:

- TO92 is clearly better than other sequences in terms of assignments.
- Although CI01 has lesser comparisons than other sequences, but it is not as good as TO92 and GB91 for most of the lengths.
- SE86 has most number of assignments compared to other sequences.
- Once again we can see the fluctuation nature of GB91. Where in some cases it has lesser assignments than TO92, for some lengths number of assignments by GB91 is more than CI01. Because of its fluctuation for different array lengths, we will not consider GB91 in further analysis.

Now we want to compare the number of swaps between Lomuto and Hoare's partitioning scheme.

Figure 10: Number of assignments for different gap sequences

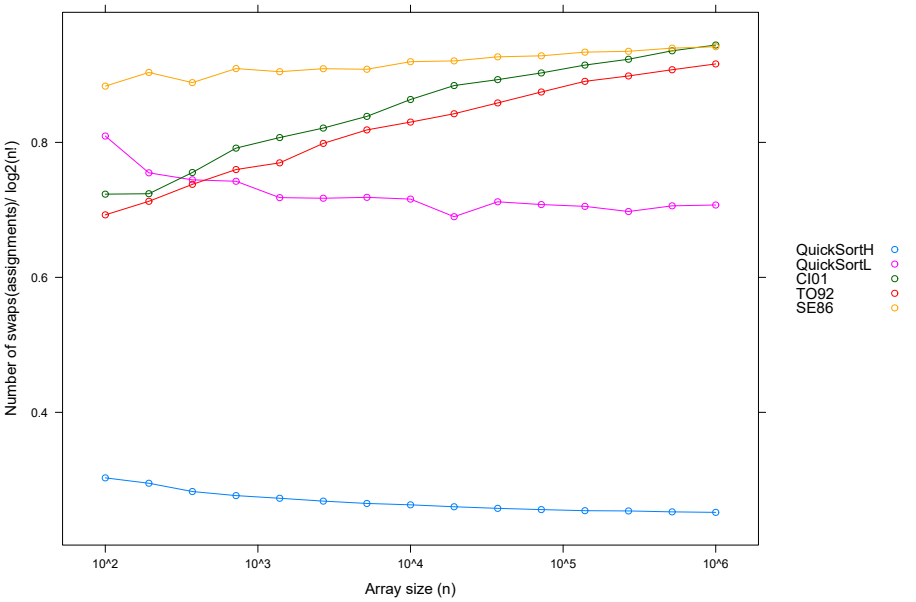


Observations:

Here we see a completely different picture from what we have seen in the number of comparisons plot. Hoare's partitioning scheme has fewer element swaps than Lomuto partitioning scheme. We would like to see between number of comparison and number of swaps, which one makes a greater impact to sort an array in less time.

Next, we will see the combined plot of Shell sorts with different gap sequences and Quick sorts with different partitioning scheme.

Figure 11: QuickSort vs ShellSort — Assignments/Swaps



Observations: Hence we conclude number of assignments of different Quick sort algorithms are much less than Shell sort algorithms with best gap sequences.

10.4 Time

We have observed efficiency where we look at the amount of time taken by different sorting methods. We would like to see how the variations in all of the plots above finally add up and which sorting algorithms are faster than the others. We will also check if Ciura's claim regarding the dominant operation in Shell sort holds true.

Figure 12: Time comparison between different gap sequences

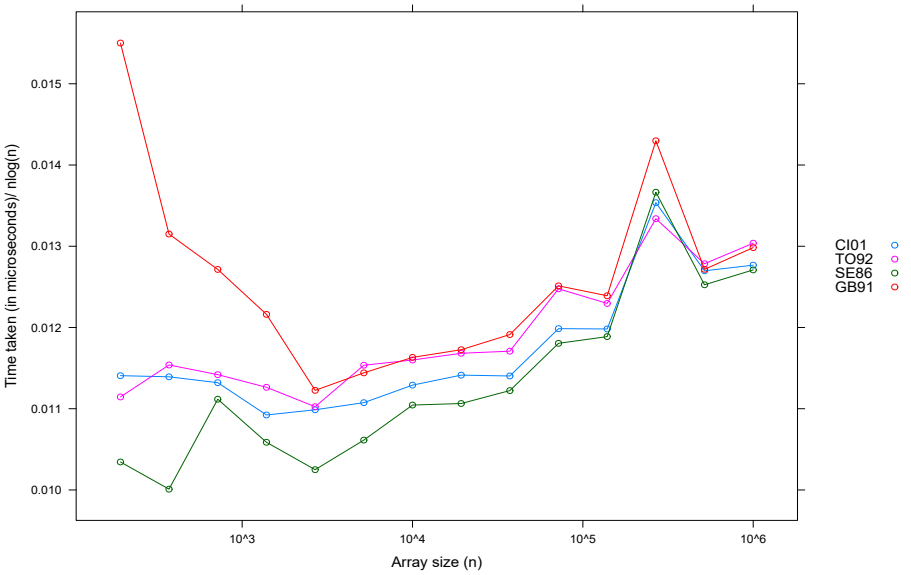
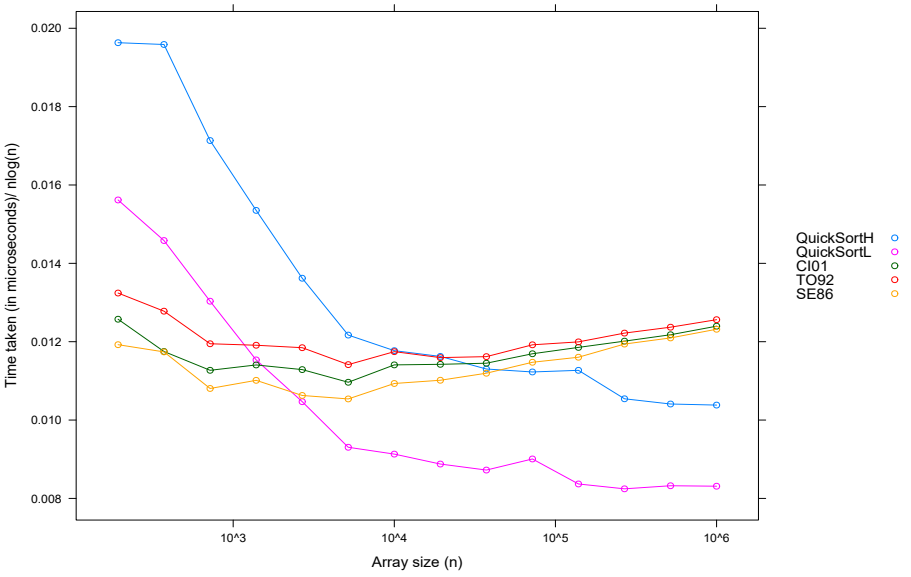


Figure 13: QuickSort vs ShellSort — Time



Observations:

- Despite performing higher number of comparisons and assignments, SE86 still turns out to be best gap sequence in the sense of taking less time for sorting an array.
- Ciura's claim regarding dominant operation in Shell sort clearly does not hold. Though CI01 has the least number of comparisons than other sequences, it fails to sort arrays faster than SE86. In fact, it comes as the 2^{nd} best in timing comparison.
- TO92 even after having less number of element assignments, takes more time than other sequences to sort.
- Shell sort algorithms seems to be a little faster for sorting arrays of length less than 10^3.
- Shell sort algorithms are faster than Quick sort algorithms implemented using Lomuto partitioning scheme for arrays of length less than 10^4

11 Limitations

- All of sorting the algorithms were written in Rcpp, hence the running time of the algorithms depend on implementation of Rcpp.
- The measurements of running time were recorded using **microbenchmark** package. So accuracy of measuring running time depends on the package.

12 Conclusions

Clearly Shell sort is not as good as Quick sort for large arrays. Again, the Quick sort algorithms in this project were implemented using only simple partitioning schemes like Lomuto and Hoare's partitioning scheme. These partitioning schemes can be improved quite a lot by choosing a better method for selecting the pivot (Some of the improved methods are ***median of three***, ***randomised pivot*** etc.). Using these improved techniques we can construct much better sorting algorithms.

However for small array sizes Shell sort performs better. As Quick sort algorithm uses recursion, it does need the small amount of extra time needed to call the recursion methods, which proves to be detrimental for small arrays. A lot of popular modern sorting algorithms hence use Insertion sort for sorting arrays of smaller sizes. It has been seen that Insertion sort is the most efficient algorithm for sorting arrays of length less than 20. Shell sort being a better variant of Insertion sort, can be used for even larger arrays.

References

[1] Wikipedia contributors, "Shellsort — wikipedia." https://en.wikipedia.org/w/index.php?title=Shellsort&oldid=939568142, 2020.

[2] Wikipedia contributors, "Coin problem — wikipedia." https://en.wikipedia.org/w/index.php?title=Coin_problem&oldid=941209878, 2020.

[3] E. S. Selmer, "On shellsort and the frobenius problem," 1987.

[4] M. Ciura, "Best increments for the average case of shellsort," in *International Symposium on Fundamentals of Computation Theory*, pp. 106–117, Springer, 2001.