Project on Introductory Computer Programming

Introduction to Shell Sort

Pritam Dey, Ayan Paul

May 4, 2020

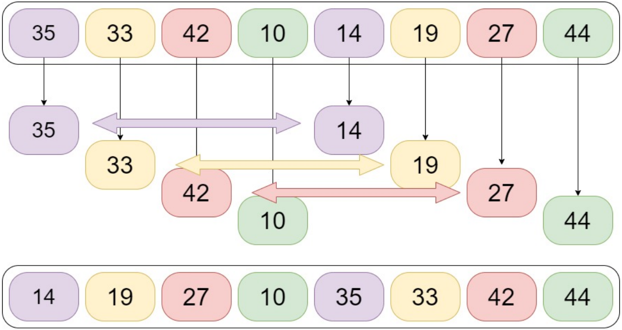
What is Shell Sort

Shell sort 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 Shell sort is to allow exchange of far items. In Shell sort, 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.

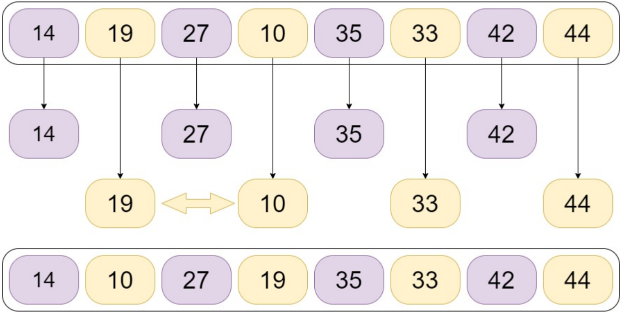
Visualisation

Figure: Sorting sub-arrays of gap 4



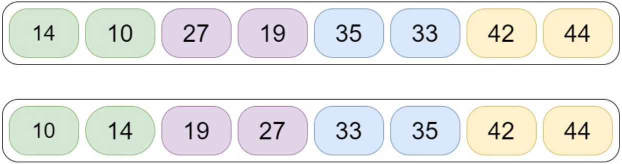
Visualisation

Figure: Sorting sub-arrays of gap 2



Visualisation

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



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. Also there is a nice property that if an array is k-sorted for some k $<$ h, then it is αh+βk sorted for any non negative integers α and β.

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 $\{4, 2, 1\}$.

There is a nice relation between Shell Sort, its gap sequences and a renowned problem in Number Theory, **The Frobenius Coin Problem**. The worst case time complexity of Shellsort can be analysed using the concept of this 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 co-primes.

Mathematical Statement of the Problem

In mathematical terms the problem can be stated as:
Given positive integers a_1,a_2,....,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 $k_1a_1+k_2a_2+....+k_na_n$,where k_1,k_2,...,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).

Frobenius Number for Small Set (of size n) 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_1 a_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 $N(a_1,a_2) = (a_1 - 1)(a_2 - 1)/2$ 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 $n = 3$ can been also determined.

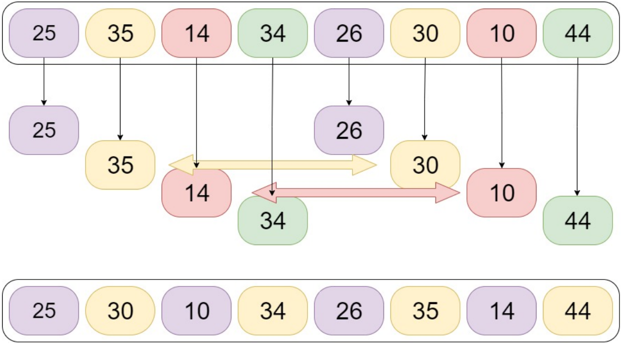
Time complexity

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

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

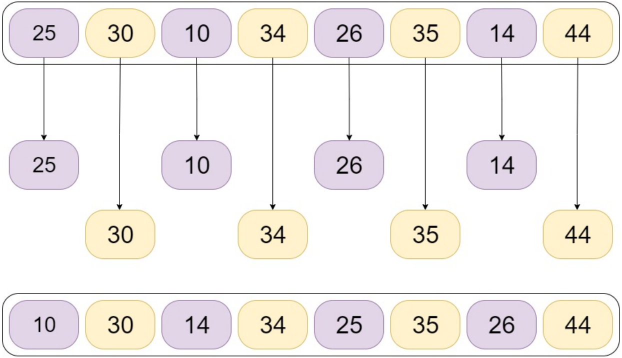
Worst case of Shell's gap sequence

Figure: Sorting sub-arrays of gap 4



Worst case of Shell's gap sequence

Figure: Sorting sub-arrays of gap 2



Gap sequences and Time complexity

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.

Properties of a *good* gap sequence

- 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.
- 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 that gap-sequence with successive ratio 2.25 is more efficient.

Properties of a *good* gap sequence

- 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.
- 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 that gap-sequence with successive ratio 2.25 is more efficient.

Properties of a *good* gap sequence

- 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.
- 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 that gap-sequence with successive ratio 2.25 is more efficient.

Properties of a *good* gap sequence

- 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

Properties of a *good* gap sequence

- 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.
- 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 that gap-sequence with successive ratio 2.25 is more efficient.

Well known gap sequences

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: $\Theta(N^{\frac{3}{2}})$

Well known gap sequences

Hibbard's Sequence

Year of Publication: 1963
General Term: $2^k - 1$
Concrete Gaps: $1, 3, 7, 15, 31, 63, \ldots.$
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: $1, 3, 5, 9, 17, 33, 65, \ldots.$
Worst Case Time Complexity: $\Theta(N^{\frac{3}{2}})$

Well known gap sequences

Pratt's Sequence

Year of Publication: 1971
General Term:
Successive numbers of the form 2^p3^q $(3 - \textit{smooth Numbers})$
Concrete Gaps: $1, 2, 3, 4, 6, 8, 9, 12, \ldots.$
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: $1, 4, 13, 40, 121, \ldots.$
Worst Case Time Complexity: $\Theta(N^{\frac{3}{2}})$

Well known gap sequences

Incerpi and Sedgewick's Sequence

Year of Publication: 1985

General Term:

$\prod_I a_q,\ where\ a_q = min\{n \in \mathbb{N} : n \geqslant (5/2)^{q+1}\ \forall p : 0 \leqslant p < q \implies gcd(a_p, n) = 1\}$

$\mathrm{I} = \{\ 0 \leqslant q < r |\ q \neq \frac{(r^2+r)}{2} - k\}$

$\mathrm{r} = \lfloor \sqrt{2k + \sqrt{2k}} \rfloor$

Concrete Gaps: $1, 3, 7, 21, 48, 112, \ldots$

Worst Case Time Complexity: $\Theta(N^{1+\sqrt{\frac{8\log(\frac{5}{2})}{\log N}}})$

Well known gap sequences

Sedgewick's First Sequence

Year of Publication: 1982
General Term: $4^k + 3.2^{k-1} + 1,$ *prefixed with* 1
Concrete Gaps: $1, 8, 23, 77, 281, \ldots.$
Worst Case Time Complexity: $\Theta(N^{\frac{4}{3}})$

Well known gap sequences

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: $1, 5, 19, 41, 109, \ldots.$

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: $\lfloor \frac{5N}{11} \rfloor, \lfloor \frac{5}{11} \lfloor \frac{5N}{11} \rfloor \rfloor, \ldots., 1$

Worst Case Time Complexity: *Unknown*

Well known gap sequences

Tokuda's Sequence

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

Ciura's sequential analysis

Ciura showed that the number of comparisons made by Shell Sort algorithm for a particular gap sequence follows approximately a normal distribution.

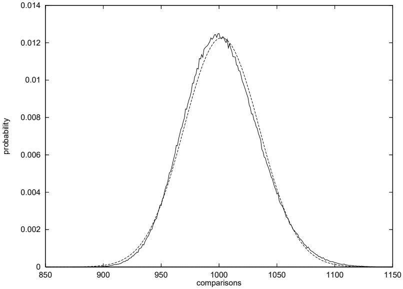


Fig. 2. Distribution of the number of comparisons in Shellsort using the sequence $(1, 4, 9, 24, 85)$ for sorting 128 elements (*solid line*), and the normal distribution with the same mean and standard devistion (*dashed line*)

Ciura's sequential analysis

- For all the arrays of a particular size, Ciura based on his experience conjectured some lower and upper cut off points θ_0 and θ_1 respectively.
- 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 $2, 3, 4, \ldots$
- With each sequence probed, he performed Shell sort on randomly generated permutations and added the number of comparisons(c_i) at each trial. He performed each test as long as

Ciura's sequential analysis

- For all the arrays of a particular size, Ciura based on his experience conjectured some lower and upper cut off points θ_0 and θ_1 respectively.
- 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 $2, 3, 4, \ldots$
- With each sequence probed, he performed Shell sort on randomly generated permutations and added the number of comparisons(c_i) at each trial. He performed each test as long as

Ciura's sequential analysis

- For all the arrays of a particular size, Ciura based on his experience conjectured some lower and upper cut off points θ_0 and θ_1 respectively.
- 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 $2, 3, 4, \ldots$
- With each sequence probed, he performed Shell sort on randomly generated permutations and added the number of comparisons(c_i) at each trial. He performed each test as long as

Sequential test

$$a_k = \frac{\sigma_{max}^2}{\theta_1 - \theta_0} ln \frac{\beta}{1-\alpha} + k\frac{\theta_0 + \theta_1}{2} \leq \sum_{i=1}^{k} c_i$$

$$r_k = \frac{\sigma_{max}^2}{\theta_1 - \theta_0} ln \frac{1-\beta}{\alpha} + k\frac{\theta_0 + \theta_1}{2} \geq \sum_{i=1}^{k} c_i$$

Here θ_0 and θ_1 are his conjectured constants for a given length of gap-sequences.

A gap sequence is considered as a good (or bad) gap sequence if average number of comparisons made by the sequence is less than θ_0 (or more than θ_1)

Dominant Operation in Shell sort

Ciura claimed 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

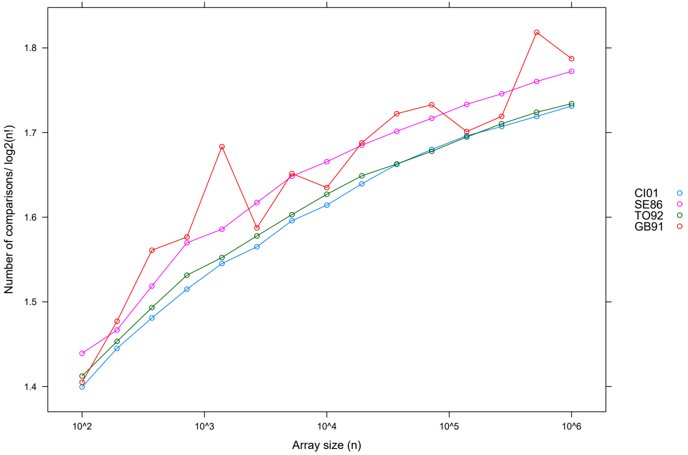
Comparing efficiency of some latest gap-sequences

We have used three different plots to compare the sorting methods. We have plotted the following variables against length of array in each plot.

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

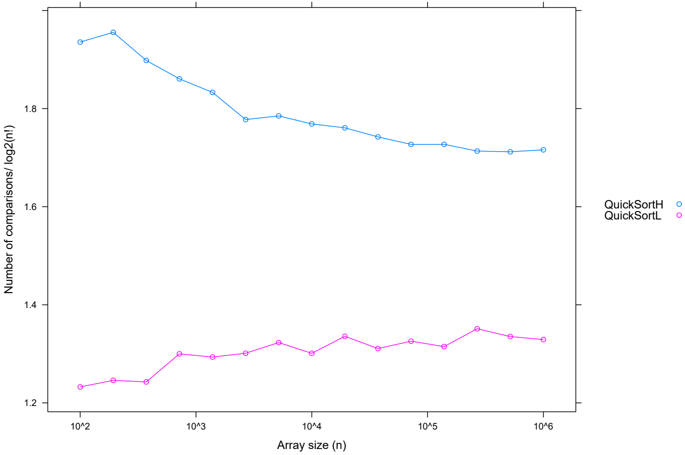
Number of comparisons

Figure: Number of comparisons for different gap sequences



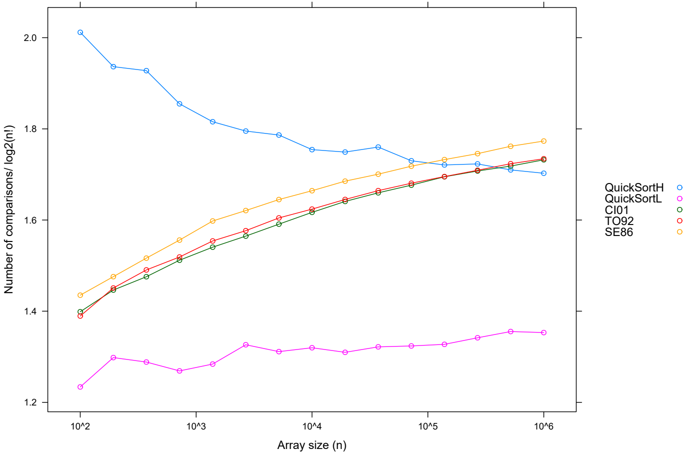
Number of comparisons

Figure: Number of comparisons for different partioning schemes



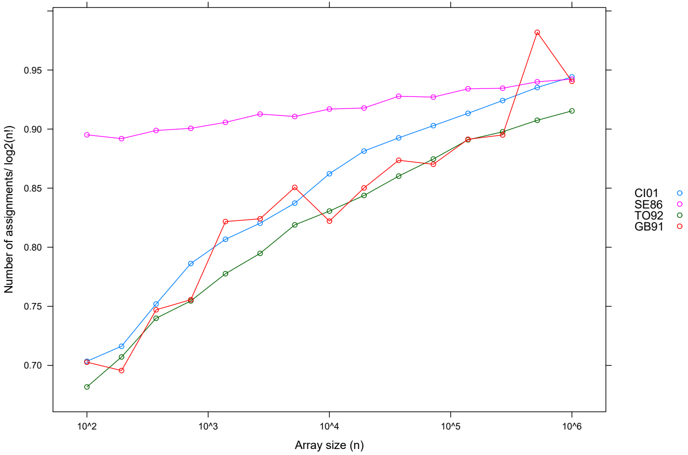
Number of comparisons

Figure: ShellSort vs QuickSort — Number of comparisons



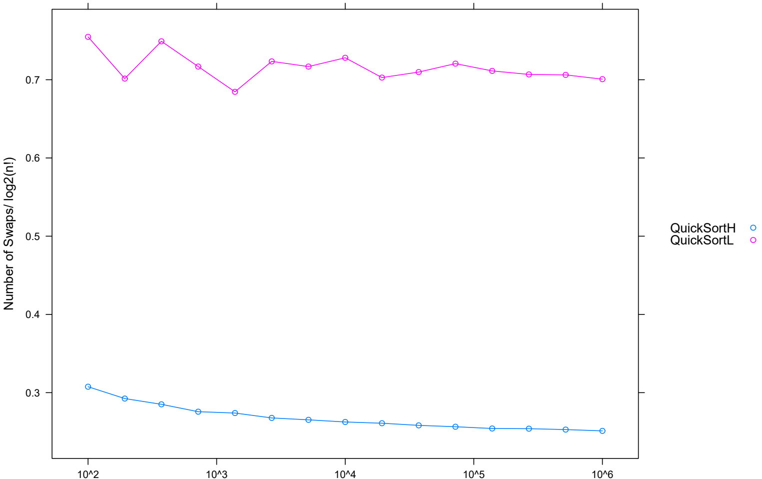
Number of Assignments/Swaps

Figure: Number of assignments for different gap sequences



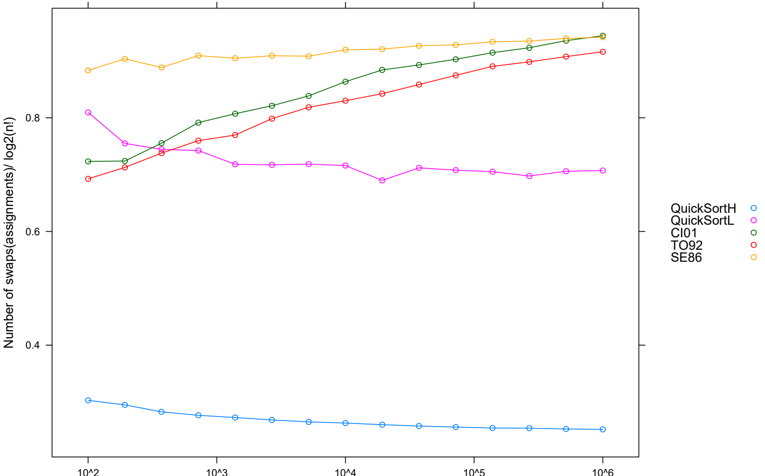
Number of Assignments/Swaps

Figure: Number of assignments for different gap sequences



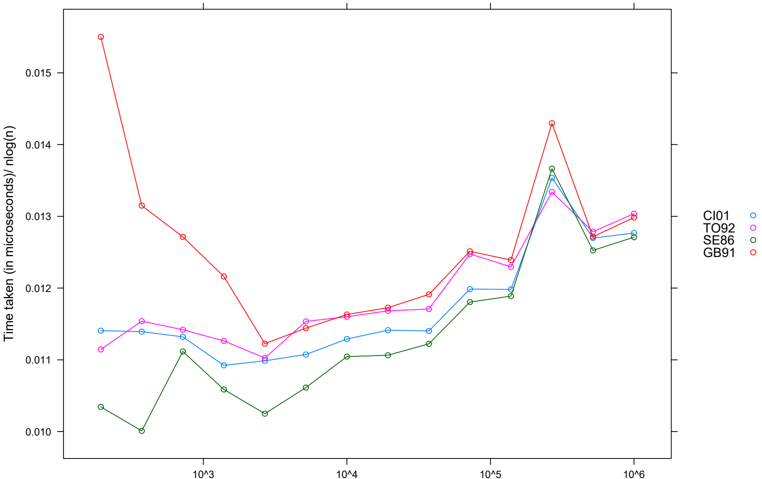
Number of Assignments/Swaps

Figure: QuickSort vs ShellSort — Assignments/Swaps



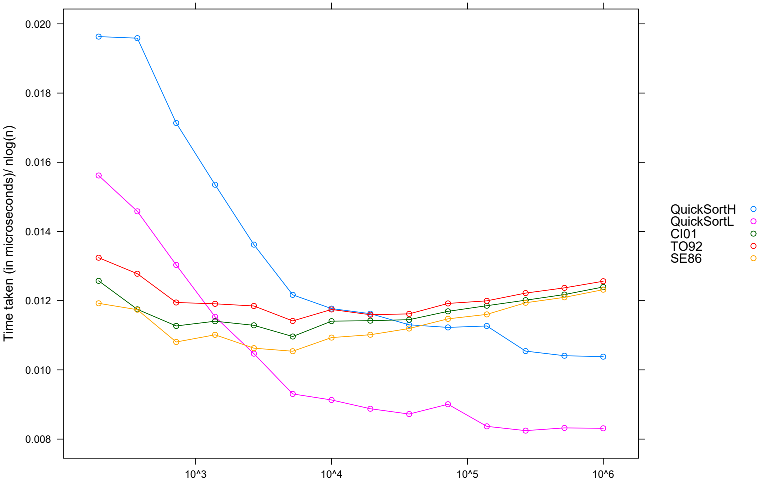
Running Time

Figure: Time comparison between different gap sequences



Running Time

Figure: QuickSort vs ShellSort — Time



Conclusions

- For smaller arrays (length $< 10^3$) Shell sort performs better than Quick sort algorithms.
- When array size increases, Quick sort becomes more efficient.
- Evidently the efficiency of insertion sort for small arrays has been extended to quite large arrays. (10^3)
- The partitioning scheme used in this project can be improved in different ways by choosing better algorithms for selecting the pivot.