CS 260P Project 1

Chad Bloxham
UID: 81321335
cbloxham@uci.edu
May 3, 2019

1 Overview

Shell sort is a generalization of insertion sort in which a "gap sequence" is chosen and the input array is iterated through once for each value in the gap sequence. Comparisons are performed between indices separated by a multiple of the gap in question. The lengths of these gaps decrease and the last gap value is 1, resulting in a standard insertion sort. This last iteration will result in fewer swaps than if we had simply done an insertion sort, since the earlier gaps have already performed a large amount of sorting. Here we examine the runtime of Shell sort for various gap sequences, both on random permutations (all permutations of $[1, N]$ are equally likely) and almost-sorted permutations ($2logN$ inversions where N is the size of the input array).

2 Algorithm

2.1 Shell Sort

The following pseudocode shows our implementation of Shell sort. Our output is a numpy array of integers corresponding to the sorted version of the input array A (which does not have to be a numpy array). We choose to use a numpy array because reading and writing an element from a numpy array is faster than doing so for a standard python list.

```
shell_sort(A, n, gaps):
    Inputs: A, an unsorted array of integers between 1 and n.
    n, the length of A.
    gaps, an array sorted in descending order of the gaps to be used for comparisons.
    Output: S, the sorted array.
    S = A
    for each gap in gaps:
        for i=gap to n-1:
            j=i
            temp = S[i]
            while j >= gap and S[j-gap] > temp:
                S[j] = S[j-gap]
                j = j - gap
            S[j] = temp
    return S
```

2.2 Computing Runtimes

We ran our Shell sort algorithm on array sizes which were powers of 2, ranging from $N_{min} = 2^6$ to $N_{max} = 2^{18}$. For each N value and for each gap sequence we generate five random permutations, time how long it takes our Shell sort algorithm to sort them (using python's *time* module), and then store the average in a numpy array. We then repeat this procedure, but now we generate almost-sorted arrays and store the average runtimes in a different numpy array. In pseudocode:

```
for each n in N:
    gap_seq = [ list containing each gap sequence outputted with this n ]
```

```
for each seq in gap_ seq:
    t = 0.0
    for i=1 to 5:
        r = random permutation of size n
        time shell_sort(r, n, seq)
        t = t + time
    store t / 5.0
    t = 0.0
    for i=1 to 5:
        as = almost-sorted permutation of size n
        time shell_sort(as, n, seq)
        t = t + time
    store t / 5.0
```

3 Results

The last gap sequence tested is one of our own invention. It is inspired by the Pratt sequence, which has been shown to have a better worst case runtime than the original gap sequence published by Shell. This may be because Shell's sequence decreases to small gap values too quickly (it starts at N/2 and repeatedly halves until it reaches 1), so the majority of sorting is done in time-intensive iterations involving smaller gaps. Pratt's sequence (all values $2^p3^q < N$ in descending order) has more large values and thus takes some of the burden off of the smaller gap values. However, Pratt's sequence is slowed down by the fact that the total number of gap values is large, resulting in more iterations overall. Our sequence is $3^p5^q < N$ in descending order. The intuition behind this choice is that it will have less total gap values than Pratt, but still retain the benefit of having more large gaps than Shell's original sequence.

3.1 Random Permutations

On random permutations, our sequence is the second best overall, only behind A036562 ($4^k+3*2^{k-1}+1 < N$ in descending order). This sequence decreases even more quickly than the original Shell sequence and its largest value is less than that of our sequence. This implies that our above intuition is not totally correct. While it is possible to improve performance by increasing the number of large gaps (as we have done), even better performance can be obtained with smaller values, and less of them. When it comes to the choice of largest gap values, it is not simply "the larger, the better". The largest gaps in A036562 are smaller and there are fewer of them, but they are able to do more sorting than the largest gap values in our sequence, taking even more burden off of the smaller gap values than ours does.

Gap Sequence	Runtime
A036562	$O(n^{1.20})$
My Seq	$O(n^{1.22})$
A168604	$O(n^{1.23})$
Pratt	$O(n^{1.25})$
Original	$O(n^{1.40})$

Figure 1: Gap sequences ranked from fastest to slowest.

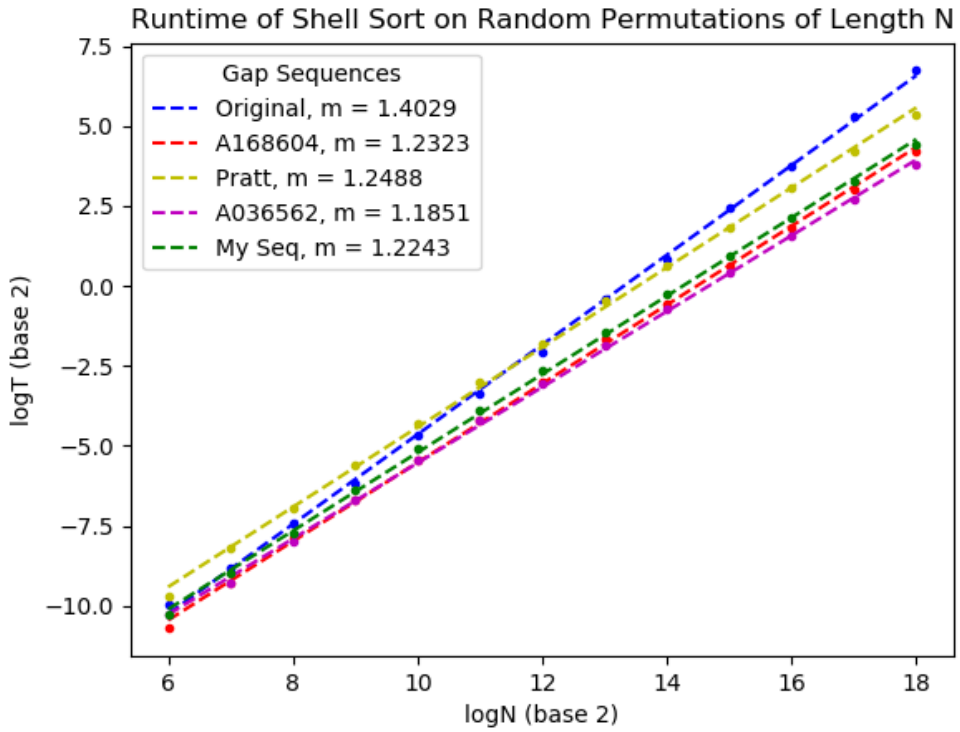

Figure 2: Shell sort performance for different gap sequences on uniformly distributed permutations.

3.2 Almost-Sorted Permutations

On permutations which are nearly sorted to begin with, Pratt and our sequence show poor performance compared to the others. This makes sense, since these sequences contain more large gap values, and these gaps are unlikely to be very helpful when the number of inversions is small. Shell's original sequence performs very well (second best overall) since it gets to the smaller gap values more quickly, and these are much more likely to sort the few existing inversions. Once again, A036562 performs the best. This is because it is able to get to the small values even more quickly than the original sequence.

Gap Sequence	Runtime
A036562	$O(n^{1.10})$
Original	$O(n^{1.16})$
A168604	$O(n^{1.16})$
My Seq	$O(n^{1.21})$
Pratt	$O(n^{1.25})$

Figure 3: Gap sequences ranked from fastest to slowest.

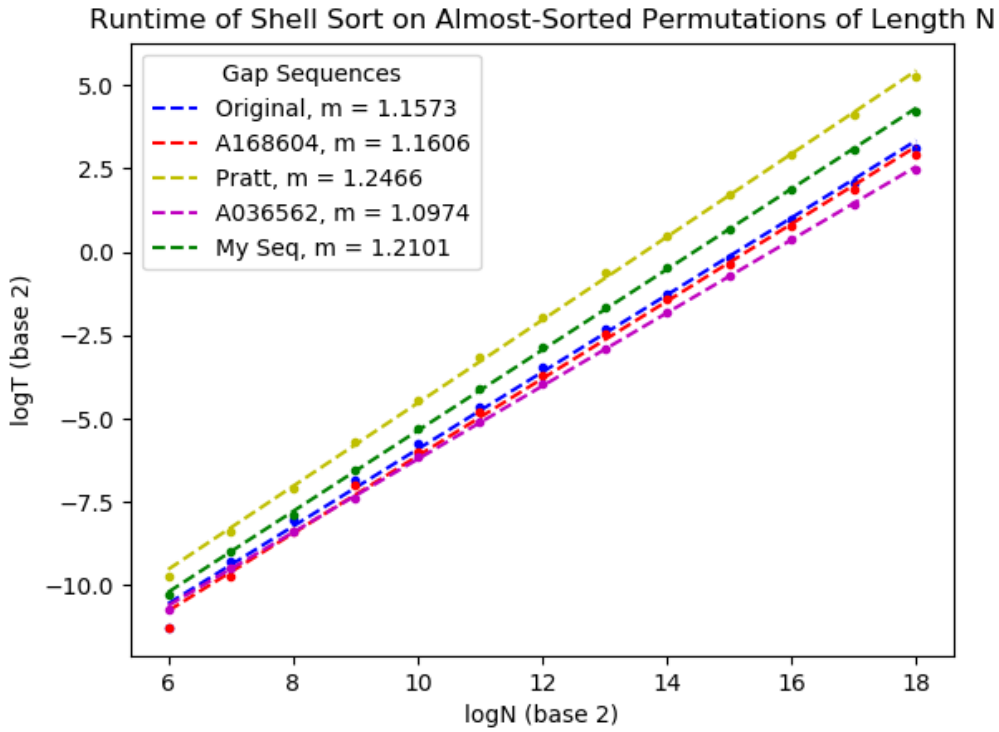

Figure 4: Shell sort performance for different gap sequences on almost-sorted permutations.