Project 1

Shell Sort

Shell Sort is an in-place comparison-based sorting. Basically, it's a generalization of sorting by exchange (bubble sort) or sorting by insertion (insertion sort). The method starts by sorting pairs of elements far apart from each other, then progressively reducing the gap between elements to be compared. Starting with far apart elements, it can move some out-of-place elements into position faster than a simple nearest neighbor exchange. The running time of a Shell-Sort algorithm depends on the gap-sequence it uses.

Pseudo-code of Shell Sort algorithm:

```
def shellsort(list, gap_sequence):
    n = len(list)
    for gap in gap_sequence:
        for i in range(gap, n):
            temp = arr[i]
            j = i
            while j >= gap and temp < arr[j - gap]:
                arr[j] = arr[j - gap]
                j -= gap
            arr[j] = temp
```

<u>Design Choices and Data Structure Used</u>

In the sorting method, we have used a dynamic array list to store the gap sequences and we loop over the gaps to sort the unsorted list for each gap. The gap sequence is generated by a method whose input is the size of the array and uses various sequence formulae to populate the gap-sequence list. The unsorted list is stored in an array of size n. We generate the unsorted list by a method whose input is again the size (n) of the array and insert n random numbers in the array. The sizes vary for different variety of test cases. We then test this randomly generated unsorted list on various shell sort methods with different gap-sequences.

<u>Different versions of Shell Sort</u>

Here are the 5 different versions of Shell Sort algorithm with different gap-sequence formulae that will be used to generate a list of gaps used in sorting (as shown above).

1. The original Shell sequence, $n/2^k$, for k=1, 2, ..., log n.
2. The sequence, 2^k-1, for k=log n, ..., 7, 3, 1.
3. The Pratt sequence, 2^p3^q, ordered from largest such number less than n down to 1.

4. The A036562 sequence, in reverse order, starting from the largest value less than n, down to 1.
5. Sequence $(1 + 3^k)/2$, in reverse order starting from the largest value less than n, down to 1.

For each sequence we'll be performing run-time experiments on 2 types of random permutations, Uniformly Distributed Permutations and Almost Sorted Permutations with multiple runs for each problem size, for increasing problem sizes.

Uniformly Distributed Permutations:

Here we randomly generate a list of un-sorted elements of a particular size and run the shell-sort algorithm for analyzing the run-time for different sizes of lists. We also perform multiple runs for a particular size after re-shuffling the elements of the list using Fisher Yates algorithm.

```
def uniformPerm(size):
        arr[size]
        for i in range(size):
                arr[i] = random(size*10)
        ret arr
```

```
def fisherYates(arr):
        n = len(arr)
        for(int i = n - 1; i > 0; i--):
                index = random(i+1)
                swap(arr[i], arr[index])
```

Almost Sorted Permutations:

Here we generate a sorted array, where the numbers in the array are from 1 to n (n is the size of the array). Then we randomly choose 2log n pairs (i,j) where i and j are random indices of the array and swap the elements at positions i and j in the array.

```
def almostSorted(size):
        arr[size]
        for i in range(size):
                arr[i] = i + 1
        ret arr
```

```
def swapNumbers(arr):
        n = len(arr)
        m = 2*log n/log 2
        for k in range(m):
                i = random(n), j = random(n)
                swap(arr[i],arr[j])
```

Experimental Analysis:

Shell Sort sequence 1:

Sequence = $n/2^k$
Where k = 1,2,…,logn

Pseudo-code:

```
def generateGapSequence(size):
    while size/2 > 0:
        gaps.add(size/2)
        size /= 2
    ret gaps
```

Uniformly Distributed Permutations:

We run the uniform distributed permutations on different sizes of the arrays and run with 5 different permutations for a particular size of array by re-shuffling the array using Fisher-Yates algorithm. We then take the average of 5 different permutations run for a particular size.

size = 16	time = 5072ns
size = 128	time = 236875ns
size = 1024	time = 10650325ns
size = 8192	time = 478649419ns
size = 65536	time = 21511575192ns

Almost Sorted Permutations:

We run the almost sorted permutations on different sizes of the arrays and run with 5 different permutations for a particular size of array by re-shuffling the sorted array by taking random 2log n pairs of indices and swapping the elements on that position. We then take the average of 5 different permutations run for a particular size.

size = 16	time = 3674ns
size = 128	time = 155209ns
size = 1024	time = 6553584ns
size = 8192	time = 276720865ns
size = 65536	time = 11684331872ns

Shell Sort sequence 2:

Sequence = $2^k - 1$
Where k = log n, … 3,2,1

Pseudo-code:

```
def generateGapSequence(size):
    temp = 2
    while temp <= size:
        gaps.add(temp-1)
        temp *= 2
    ret gaps
```

Uniformly Distributed Permutations:

We run the uniform distributed permutations on different sizes of the arrays and run with 5 different permutations for a particular size of array by re-shuffling the array using Fisher-Yates algorithm. We then take the average of 5 different permutations run for a particular size.

size = 16	time = 2192ns
size = 128	time = 56188ns
size = 1024	time = 1440343ns
size = 8192	time = 36922092ns
size = 65536	time = 946469578ns

Almost Sorted Permutations:

We run the almost sorted permutations on different sizes of the arrays and run with 5 different permutations for a particular size of array by re-shuffling the sorted array by taking random 2log n pairs of indices and swapping the elements on that position. We then take the average of 5 different permutations run for a particular size.

size = 16	time = 2056ns
size = 128	time = 50268ns
size = 1024	time = 1334567ns
size = 8192	time = 35858706ns
size = 65536	time = 943658204ns

Shell Sort sequence 3:

Sequence = 2^p3^q (Pratt Sequence)
Ordered from largest such number less than n down to 1

Pseudo-code:

```
def generateGapSequence(size):
    for(p = 0; p < n; p++):
        pp = 2^p
        if pp >= size: break
        for(int q = 0; q < n; q++):
            pq = pp*3^q
            if pq >= size: break
            gaps.add(pq)
    reversesort(gaps)
    ret gaps
```

Uniformly Distributed Permutations:

We run the uniform distributed permutations on different sizes of the arrays and run with 5 different permutations for a particular size of array by re-shuffling the array using Fisher-Yates algorithm. We then take the average of 5 different permutations run for a particular size.

size = 16	time = 4539ns
size = 128	time = 141936ns
size = 1024	time = 3080831ns
size = 8192	time = 66871470ns
size = 65536	time = 1451490015ns

Almost Sorted Permutations:

We run the almost sorted permutations on different sizes of the arrays and run with 5 different permutations for a particular size of array by re-shuffling the sorted array by taking random 2log n pairs of indices and swapping the elements on that position. We then take the average of 5 different permutations run for a particular size.

size = 16	time = 3290ns
size = 128	time = 87487ns
size = 1024	time = 1784126ns
size = 8192	time = 36383639ns
size = 65536	time = 741970476ns

Shell Sort sequence 4:

Sequence = $4^{k+1} + 3*2^{k} + 1$
Ordered from the largest value less than n, down to 1

Pseudo-code:

```
def generateGapSequence(size):
    p = 1, m = 0
    while p < size:
        gaps.add(p)
        p = 4^(m+1) + 3*2^m + 1
        m++
    reversesort(gaps)
    ret gaps
```

Uniformly Distributed Permutations:

We run the uniform distributed permutations on different sizes of the arrays and run with 5 different permutations for a particular size of array by re-shuffling the array using Fisher-Yates algorithm. We then take the average of 5 different permutations run for a particular size.

size = 16	time = 1686ns
size = 128	time = 33005ns
size = 1024	time = 645478ns
size = 8192	time = 12626924ns
size = 65536	time = 247010991ns

Almost Sorted Permutations:

We run the almost sorted permutations on different sizes of the arrays and run with 5 different permutations for a particular size of array by re-shuffling the sorted array by taking random 2log n pairs of indices and swapping the elements on that position. We then take the average of 5 different permutations run for a particular size.

size = 16	time = 1159ns
size = 128	time = 22684ns
size = 1024	time = 443763ns
size = 8192	time = 8681015ns
size = 65536	time = 169820068ns

Shell Sort sequence 5:

Sequence = $(1 + 3^k)/2$
Ordered from largest such number less than n down to 1

Pseudo-code:

```
def generateGapSequence(size):
    p = 1, m = 1
    while p < size:
        gaps.add(p)
        p = (1 + 3^m)/2
        m++
    reversesort(gaps)
    ret gaps
```

Uniformly Distributed Permutations:

We run the uniform distributed permutations on different sizes of the arrays and run with 5 different permutations for a particular size of array by re-shuffling the array using Fisher-Yates algorithm. We then take the average of 5 different permutations run for a particular size.

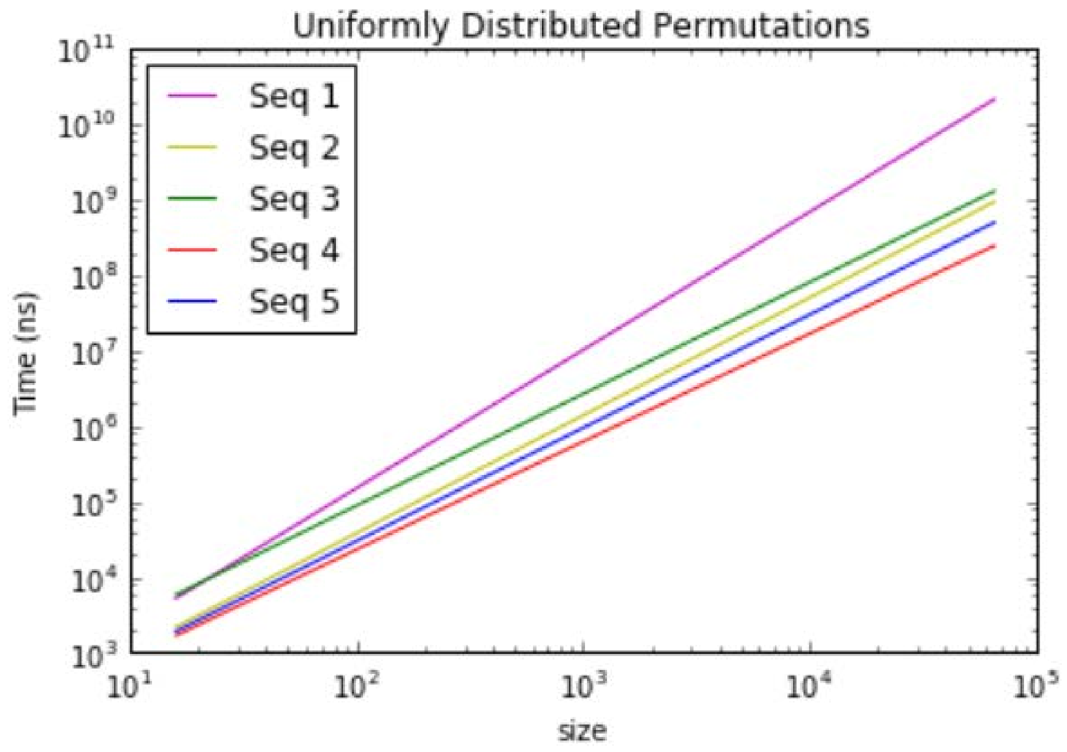
size = 16	time = 1920ns
size = 128	time = 43445ns
size = 1024	time = 983940ns
size = 8192	time = 22243756ns
size = 65536	time = 503316890ns

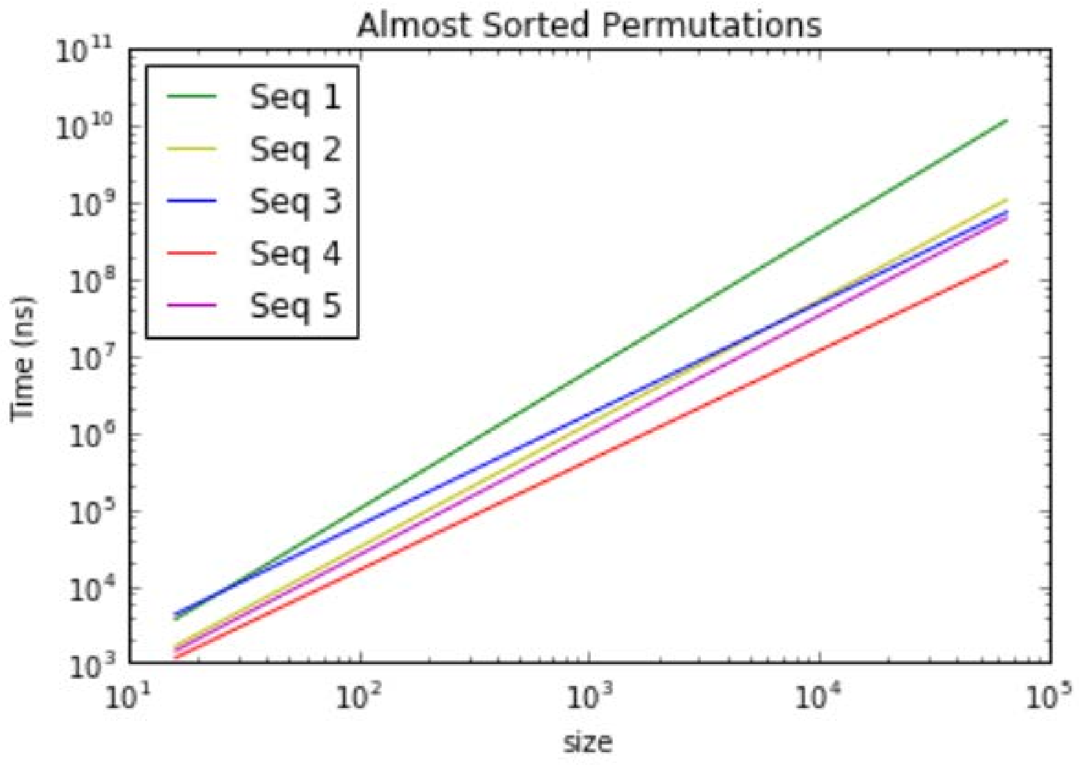
Almost Sorted Permutations:

We run the almost sorted permutations on different sizes of the arrays and run with 5 different permutations for a particular size of array by re-shuffling the sorted array by taking random 2log n pairs of indices and swapping the elements on that position. We then take the average of 5 different permutations run for a particular size.

size = 16	time = 1436ns
size = 128	time = 36831ns
size = 1024	time = 943673ns
size = 8192	time = 21490336ns
size = 65536	time = 480100801ns

Graphs:





Observations:

The slope of the graph will determine the complexity of the algorithm. The slope gives us the polynomial power of the complexity.

For example:

For sequence 1: (1024, 10650325) and (8192, 478649419)

$$m = \frac{\log Y2 - \log Y1}{\log X2 - \log X1}$$

$$m = \frac{\log Y2/Y1}{\log X2/X1}$$

$$m = \frac{\log 478649419/10650325}{\log 8192/1024}$$

$$m = \frac{1.652655}{0.90309}$$

$$m = 1.83$$

So, the complexity of sequence 1, i.e. $n/2^k$ is $O(n^{1.83})$

Similarly, after calculating the slope of other lines in the graphs, following are the complexities of all the shell sort algorithms:

1. Sequence 1: $n/2^k = O(n^{1.83})$
2. Sequence 2: $2^k - 1 = O(n^{1.56})$
3. Sequence 3: $2^p3^q = O(n^{1.48})$
4. Sequence 4: $4^{k+1} + 3.2^k + 1 = O(n^{1.43})$
5. Sequence 5: $(3^k + 1)/2 = O(n^{1.50})$

Conclusion:

Therefore, the sequence order according to their complexity from worst case to best case is:

Seq 1 [$O(n^{1.83})$] > **Seq 2** [$O(n^{1.56})$] > **Seq 5** [$O(n^{1.50})$] > **Seq 3** [$O(n^{1.48})$] > **Seq 4** [$O(n^{1.43})$]