Karan Khosla **91725095**

CompSci 260P Project #1- Shell Sort

INTRODUCTION

Shell Sort is a modified version of insertion sort where we start with a gap sequence and sort the elements at those gaps. Then we reduce the gap sequence using some formula, till we reach gap sequence of 1(which is the insertion sort). The main idea is that when we reach a gap sequence of 1, most of the elements of the array would be in a sorted order, hence the time complexity is expected to be less than insertion sort(O(n^2)).

Example :
Say we are given an initial sequence of elements in the array:

Initial array : 324 529 10 582 919 120 4 13 84 30
The gap sequence is as follows : 5, 2,1
Lets see how shell sort would work. At gap sequence of 5, it would sort elements at a gap of 5. So, after that iteration, our array would look like

120 4 10 84 30 324 529 13 582 919

Then the gap sequence becomes 3.
After that iteration, our array would look like:

84 4 10 120 13 324 529 30 582 919

Lastly the array would look like

4 10 13 30 84 120 324 529 582 919(Sorted array).

DESIGN CHOICES IN MY IMPLEMENTATION

I calculate the time to shell sort for different number of elements from 10 to 1000000. I use a for-loop for a particular value of n, and for each value, I take the average of the running time of 10 instances of the input array and think of it as the time taken at that value of n.

Data Structures used-: All my shell sort implementations use 2 vectors for storing the uniform distribution and almost sorted distribution respectively. One vector is used to store the gapSequence. And two other vectors store the corresponding times for different values of input size.

Functions used-:
1. generateUnifDist(int n)
I generate a uniform distributed sequence of numbers of length n using the random library of C++ (uniform_int_distribution).

2. generateGapSequence(int n)
This method is used to generate the gap sequence of the particular shell sort to be used in our sorting algo.

3.fyShuffle(vector<int> &dist)
This method employs the fischer-yates shuffle on the input array and returns a random sequence of integers to be used during the multiple runs for the same input size. I use this function to shuffle the uniform distribution sequence after it is sorted once to get an average over 10 run times for each value of n.

Pseudocode of FYShuffle():

```
For i from n - 1 downto 1:
    j <- rand() % (i + 1)
    swap(arr[i],arr[j])
```

4.generateAlmSortPerm(int n)
Similar to 1 but generates an almost sorted array of numbers.

5.shuffle(int n)
Similar to 3, but used to shuffle the sorted sequence by picking 2*logn pairs to get an almost sorted sequence.

Pseudocode for shuffle:

```
for i from 0 to 2*logn:
    el1 <- rand() % n
    el2 <- rand() % n
    swap(arr[el1],arr[el2])
```

Pseudocode of the ShellSort function:

```
for k from 0 to gapSequence.size():
    gap <- gapSequence[k]
    for i from gap to n-1:
        int temp <- dist[i]
        int j <- i
        while (j >= gap && temp < dist[j - gap])
            dist[j] <- dist[j - gap]
            j -= gap
        dist[j] <- temp
```

Apart from the 4 gap sequence, I chose to use Knuth's gap sequence, which is generated using the formula (3^k - 1)/2.

Gap Sequences

1. Original shellsort with $n/2^k$ as the gap sequence, for k=1,2,...,log n

```
Pseudocode for generating gapSequence :
    for int i from n/2 to 1:
        gapSequence.add(i)
        n<-n/2
```

2. Hibbard sequence 2^k-1

```
Pseudocode for generating gapSequence :
    for k from 1 to $2^k$-1 < n
        gapSequence.add($2^k$-1)
        k++
    gapSequence.reverse()
```

3. Pratt Sequence 2^p3^q

```
Pseudocode for generating gapSequence :
    twoIndex <- 0, threeIndex <- 0
    gapSequence[0] <- 1
    i = 0
    while gapSequence[i] < n
        If gapSequence[twoIndex] * 2 < gapSequence[threeIndex] * 3
            gapSequence.add(gapSequence[twoIndex++] * 2)
        else  gapSequence[twoIndex] * 2 > gapSequence[threeIndex] * 3
            gapSequence.add(gapSequence[threeIndex++] * 3)
        else
            gapSequence.add(gapSequence[twoIndex++] * 2)
            threeIndex++;
        i++
    gapSequence.reverse()
```

4. A036562 seqeunce

```
Pseudocode for generating gapSequence :
    i <- 0
    gapSequence.add(1)
    while (4^(i+1) + 3*2^i + 1 < n)
        gapSequence.add(4^(i+1) + 3*2^i + 1)
        i++
    gapSequence.reverse()
```

5. Knuth sequence (3^(k) - 1)/2

```
Pseudocode for generating gapSequence :
    i <- 0
    gapSequence.add(1)
    while (4^(i+1) + 3*2^i + 1 < n)
        gapSequence.add(4^(i+1) + 3*2^i + 1)
        i++
gapSequence.reverse()
```

Running time for uniform distribution sequence in nanoseconds
(Taken over running the sorting procedure 10 times for each value of n and then taking an average)

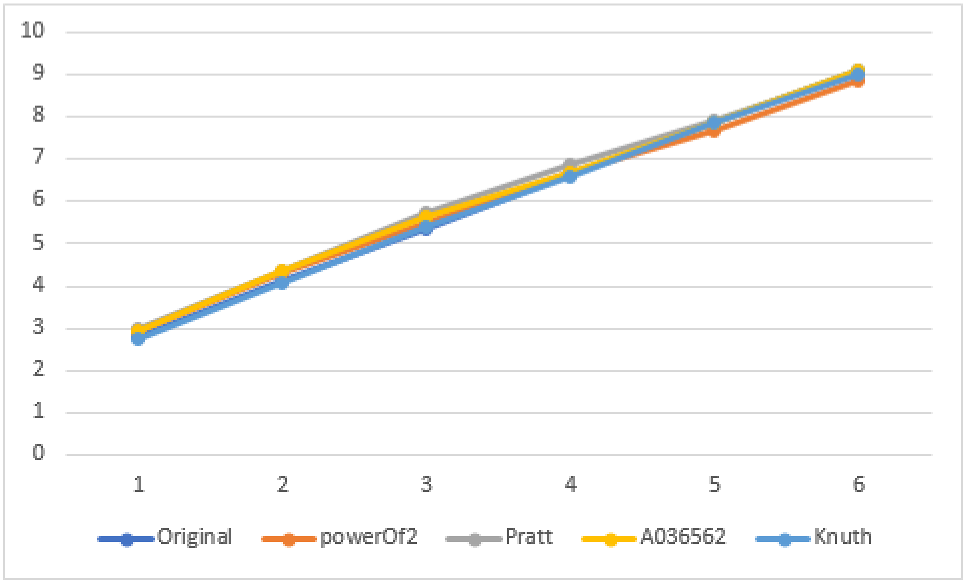
n	Original	Hibbard	Pratt	A036562	Knuth
10	634.9	881.2	916.9	846.5	528.8
100	13011.7	19641	23131.9	22884.8	12447.5
1000	225146.1	314887.3	516267	430404.7	250816.8
10000	4102878.3	4723872.7	7852891.9	4749225.7	3820854.9
100000	71915663.7	50345529.7	87166000.9	71869344	74435826.5
1000000	1242207793	731132416.3	1175800512	1289596028	1039818353

Running time for almost sorted sequence in nanoseconds

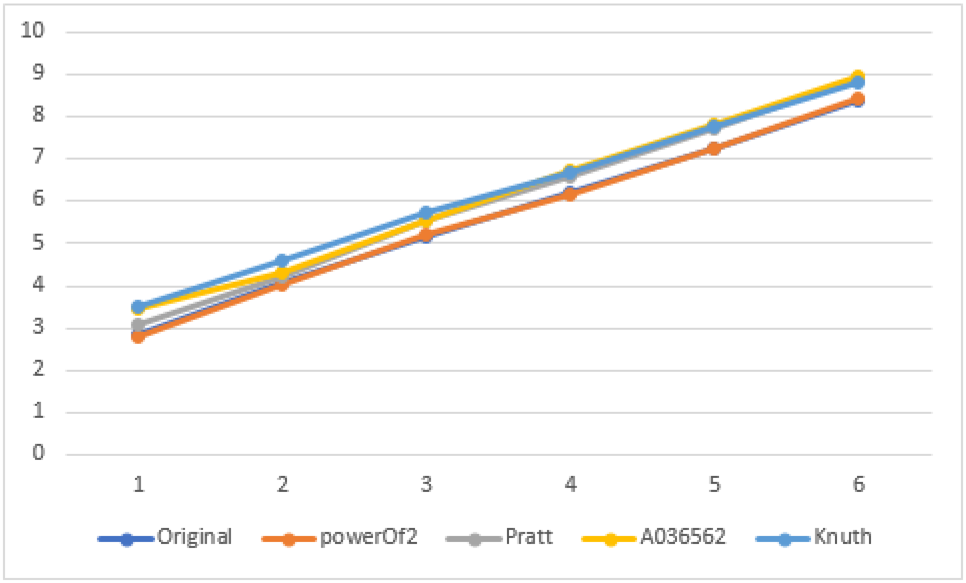
n	Original	Hibbard	Pratt	A036562	Knuth
10	670.1	634.6	1146.4	2820.9	3173.4
100	11742	10402.3	17278.3	20945.6	40727.4
1000	149721.4	156914.8	341333.4	343625.7	549201.3
10000	1567489.4	1477506.3	3976452.4	5445516.6	4993448.4
100000	18388547.4	17375940	53415401.7	64162581.8	57485465
1000000	258435317.2	284999922.5	803515664.8	883664257.6	681260966.6

Experimental analysis for different gap sequences

log log plot for uniform distribution-:



log log plot for almost sorted sequence-:



Calculating slopes of the log log plot to determine the relative running times-:

1. **Original sequence -:**
    a. **Slope for uniform distribution-:** 1.237371
    b. **Slope for almost sorted sequence -:** 1.147804
2. **Hibbard sequence**

   a. **Slope for uniform distribution-:** 1.162035
   b. **Slope for almost sorted sequence -:** 1.114896

3. **Pratt sequence**
   a. **Slope for uniform distribution-:** 1.12998652
   b. **Slope for almost sorted sequence -:** 1.10732785
4. **A036562 sequence**
   a. **Slope for uniform distribution-:** 1.133910006
   b. **Slope for almost sorted sequence -:** 1.139005458

5. **Knuth sequence**
   a. **Slope for uniform distribution-:** 1.145175
   b. **Slope for almost sorted sequence -:** 1.073755

Slope of the log-log plot signifies the exponent of the n term in the time complexity. Looking at the slopes for only the uniform distribution sequence, we can determine the order of the sorting times for the 5 sequences from **worst to the best**.

Sequence 1-: Original shell sequence $n/2^k$ - 1.237371
Sequence 2-: Hibbard sequence 2^k-1 - 1.162035
Sequence 5-: Knuth sequence $(3^k-1)/2$ - 1.145175
Sequence 4-: A036562 sequence - 1.133910006
Sequence 3-:Pratt Sequence 2^p3^q - 1.12998652