CSCI205 Computer Science III/Data Structures
Lab Assignment
Comparison vs Non-comparison Sorts

Shellsort is an optimization of insertion sort that allows the exchange of items that are far apart. The idea is to arrange the list of elements so that, starting anywhere, taking every h^{th} element produces a sorted list. Such a list is said to be ***h-sorted***. It can also be thought of as *h* interleaved lists, each individually sorted. Beginning with large values of *h* 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

Examples:

Shell Sort, K = 5: Perform an Insertion Sort on items that are K distance apart. This allows for swaps of larger distances than the traditional Insertion Sort.

Pass	List (K=5)										Notes
1	77	62	14	9	30	21	80	25	70	55	Swap
	21	62	14	9	30	77	80	25	70	55	In order
	21	62	14	9	30	77	80	25	70	55	In order
	21	62	14	9	30	77	80	25	70	55	In order
	21	62	14	9	30	77	80	25	70	55	In order

Shell Sort, K = 2: Decrease K and repeat Insertion Sort on items that are K distance apart.

Pass	List (K=2)										Notes
2	21	62	14	9	30	77	80	25	70	55	Swap
	14	62	21	9	30	77	80	25	70	55	Swap
	14	9	21	62	30	77	80	25	70	55	In order
	14	9	21	62	30	77	80	25	70	55	In order
	14	9	21	62	30	77	80	25	70	55	In order
	14	9	21	62	30	77	80	25	70	55	Swap
	14	9	21	62	30	25	80	77	70	55	Swap
	14	9	21	25	30	62	80	77	70	55	In order
	14	9	21	25	30	62	80	77	70	55	Swap
	14	9	21	25	30	62	70	77	80	55	In order
	14	9	21	25	30	62	70	77	80	55	Swap
	14	9	21	25	30	62	70	55	80	77	Swap
	14	9	21	25	30	55	70	62	80	77	In order

Shell Sort, K = 1: Decrease K and repeat Insertion Sort on items that are K distance apart. The gap sequence must terminate with a **1- sort** . . . this is the traditional Insertion Sort. You should notice that when 1-sorting, the items being considered have very short distances to cover before finding their final resting spot.

Pass	List (K=1)										Notes
3	14	9	21	25	30	55	70	62	80	77	Swap
	9	14	21	25	30	55	70	62	80	77	In order
	9	14	21	25	30	55	70	62	80	77	In order
	9	14	21	25	30	55	70	62	80	77	In order
	9	14	21	25	30	55	70	62	80	77	In order
	9	14	21	25	30	55	70	62	80	77	In order
	9	14	21	25	30	55	70	62	80	77	Swap
	9	14	21	25	30	55	62	70	80	77	In order
	9	14	21	25	30	55	62	70	80	77	In order
	9	14	21	25	30	55	62	70	80	77	Swap
	9	14	21	25	30	55	62	70	77	80	In order

Pseudo Code using Marcin Ciura's gap sequence

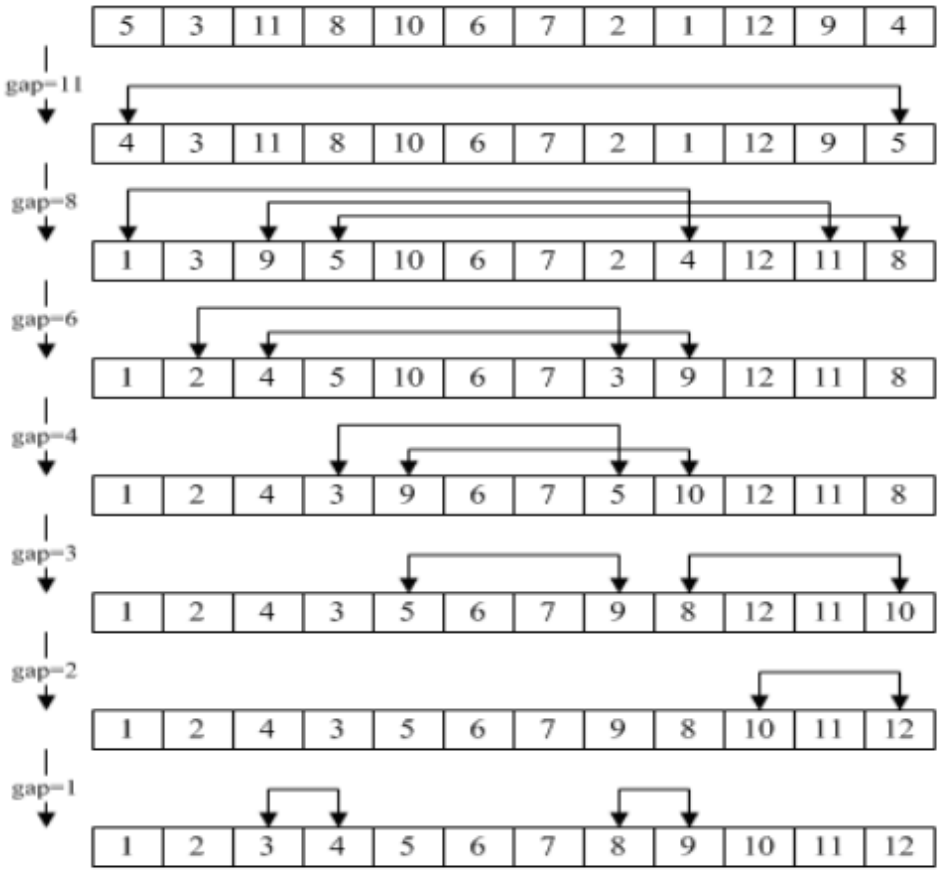
```
# Sort an array a[0...n-1].
gaps = [701, 301, 132, 57, 23, 10, 4, 1]

# Start with the largest gap and work down to a gap of 1
foreach (gap in gaps)
{
    # Do a gapped insertion sort for this gap size.
    # The first gap elements a[0..gap-1] are already in gapped order
    # keep adding one more element until the entire array is gap sorted
    for (i = gap; i < n; i += 1)
    {
        # add a[i] to the elements that have been gap sorted
        # save a[i] in temp and make a hole at position i
        temp = a[i]
        # shift earlier gap-sorted elements up until the correct location for a[i] is found
        for (j = i; j >= gap and a[j - gap] > temp; j -= gap)
        {
            a[j] = a[j - gap]
        }
        # put temp (the original a[i]) in its correct location
        a[j] = temp
    }
}
```

The Comb Sort: This “gap sorting” approach is also the basis for an improvement on the Bubble Sort called the Comb Sort.

The basic idea is to eliminate **turtles**, or small values near the end of the list, since in a bubble sort these slow the sorting down tremendously. **Rabbits**, large values around the beginning of the list, do not pose a problem in bubble sort.

The inner loop of bubble sort, which does the actual swap, is modified such that the gap between swapped elements goes down (for each iteration of the outer loop) in steps of a **"shrink factor”**. The gap starts out as the length of the list ***n*** being divided by the shrink factor ***k*** (generally 1.3; see below) and one pass of the aforementioned modified bubble sort is applied with that gap. Then the gap is divided by the shrink factor again, the list is sorted with this new gap, and the process repeats until the gap is 1. At this point, comb sort continues using a gap of 1 until the list is fully sorted. The final stage of the sort is thus equivalent to a bubble sort, but by this time most turtles have been dealt with, so a bubble sort will be efficient

The shrink factor has a great effect on the efficiency of comb sort. **k = 1.3** has been suggested as an ideal shrink factor after empirical testing on over 200,000 random lists. A value too small slows the algorithm down by making unnecessarily many comparisons, whereas a value too large fails to effectively deal with turtles, making it require many passes when gap size = 1.



```
function combsort(array input)

   gap := input.size // Initialize gap size
   shrink := 1.3 // Set the gap shrink factor
   sorted := false

   loop while sorted = false
       // Update the gap value for a next comb
       gap := floor(gap / shrink)
       if gap > 1
           sorted := false // We are never sorted as long as gap > 1
       else
           gap := 1
           sorted := true // If there are no swaps this pass, we are done
       end if

       // A single "comb" over the input list
       i := 0
       loop while i + gap < input.size // See Shell sort for a similar idea
           if input[i] > input[i+gap]
               swap(input[i], input[i+gap])
               sorted := false
               // If this assignment never happens within the loop,
               // then there have been no swaps and the list is sorted.
            end if

            i := i + 1
        end loop


    end loop
end function
```

1. Write a function called **int combSort(vector<int> list)** that applies the comb sort algorithm with a shrink factor of 1.3 and returns the number of swaps

2. Write a function called **int shellSort(vector<int> list, vector<int> sequence)** that applies the shell sort algorithm to **array** using h values from the **sequence** array and returns the number of swaps or copies . . . however you want to view the inherent operation.

3. **Shell Sort Gap: <u>Hibbard Sequence:</u>**

    a. h = 1
    b. (2 ^ h) – 1
    c. h++
    d. Write a function called **vector<int> hibbard(int size)** that implements the Hibbard Sequence algorithm and returns an array of the sequence, based on **size.** **NOTE:** I want you to write logic that creates the sequence . . . not simply copy it.

4. **Shell Sort Gap: <u>Sedgwick Sequence:</u>** Interleave the following two sets

    a. **Set One (1, 19, 109, 505, 2161 . . .):**
        i. h = 0
        ii. 9(4 ^ h – 2 ^ h) + 1
        iii. h++

    b. **Set Two (5, 41, 209, 929, 3905 . . .):**
        i. h = 0
        ii. 2 ^ (h + 2)(2 ^ (h + 2) – 3) + 1
        iii. h++

    c. Write a function called **vector<int> sedgwick(int size)** that implements the Sedgwick Sequence algorithm and returns a vector of the sequence based on **size.** **NOTE:** I want you to write logic that creates the sequence . . . not simply copy it. You will need to create each sequence separately and then interleave them.

5. **Shell Sort Gap: <u>The Knuth Sequence:</u>** h = h * 3 + 1

    a. You have been given this function in this weeks repo pull

6. In this week's code you have a function called **vector<int> generate_vector(int size, char type)** that creates and returns a vector of random numbers with **size** values. **type** will specify one of the following.

    a) 'r' = random
    b) 'a' = ascending sorted
    c) 'd' = descending sorted
    d) 'p' = partially sorted

8. **main** should

    a. Run each of the algorithms with various sizes of the array and different types of arrays (random, inverse, partial sort).

9. Once you have gathered all data from your program plot the data using your Python utility. Be sure that you have run the algorithms enough times to create an informative graph depicting how each algorithm/gap sequence responds to the varying input. You may want to create multiple graphs: One for size, one for each type of array. **Your graphs must be clearly labeled.** Here are the graphs I want to see

    1. **Gap Sorts:** Comb with a shrink factor of 1.3 and Shell sort with the gap sequences listed above.
    2. **Bubbly Sorts:** Comb (1.3) and Regular Bubble

Non-Comparison Sorts: The following sorting algorithms sort without comparing items. Implement the following algorithms, demonstrate that they work and plot their efficiency.

In order to accomplish the ordering without comparisons, significant memory needs to be allocated. For each of the following sorts design experiments to graph their efficiency in terms of

- **Operations:** array assignments will be of interest here
- **Auxilliary memory:** additional memory needs to be allocated
- **Big O:** Radix sorts operates in O(*nk*) time, where *n* is the number of keys, and *k* is the key length(width). K = $\log_{base}$(longest number). Count the relevant operations and see if your algorithm conforms.

Counting Sort:

- Efficient when your key value range (range of values to be sorted) and quantity is relatively small.
- You will need two additional arrays of a size that represents the range of the key values. So, if your values range from 1 - 100 then your workspace arrays need to have an index of 100
- **Histogram:** Loop through the original array counting the number of instances of identical keys. Increment the indices in the extra array that correspond to the **original key value**. This concept is called a histogram.
- **Prefix Sum:** Loop through the extra array and compute a "prefix sum" on the counts of objects. This allows you to compute the index range of the objects final resting place. A prefix sum is similar to the Fibonacci sequence and consists of the algorithm:

  y0 = x0
  y1 = x0 + x1
  y2 = x0 + x1 + x2
  y3 = x0 + x1 + x2 + x3
  etc . . . etc . . .

  If your object counts are: 3, 6, 2, 8, 1, 2, 4 The corresponding prefix sum is: 3, 9, 11, 19, 20, 22, 26. Store the prefix sum in the extra array over-writing the object counts.

Index	0	1	2	3	4	5	6
Histogram	3	6	2	8	1	2	4
Prefix Sum	3	9	11	19	20	22	26

- Do a **reverse pass** over the **original array** beginning at array[length – 1]. The key value in this location represents the index of the **prefix sum value** in the extra array.
- Access the extra array at **extraArray[array[length - 1]]** and decrement it's value by 1.
- This decremented value will represent the index of the final resting place of the value at **array[length - 1]**.
- Copy this value into a third array at

  thirdArray[extraArray [array[nElems - 1]]] = array[nElems - 1]

Repeat until you have reached array[0]

- View a visualization of this algorithm here: **http://www.cs.usfca.edu/~galles/visualization/CountingSort.html**

Radix Sort:

- For this sort you arrange key values in multiple passes based on each of their individual digits starting with the LSD (least significant digit)
- You will need an additional array of identical size and an additional array of the size of your **radix** (number system base . . .
    - Base 2 would need an array of size 2;
    - Base 10 would need an array of size 10;
    - Base 16 would need an array size of 16 . . . etc)
- Loop through the original array accessing the LSD of each individual value . . . value % 10
- Perform a counting sort on these individual digits identical to above. When the counting sort is completed for the LSD write the sorted values back into the original array at their new positions
- Start the process again, only this time access the digit to the right of the LSD . . . (value % 100) / 10
- Repeat accessing each individual digit (next would be . . . (value % 1000) / 100)
- View a visualization of this algorithm here: **http://www.cs.usfca.edu/~galles/visualization/RadixSort.html**

10. All code must be error, warning and leak free. Submit your C++ files, and graphs.