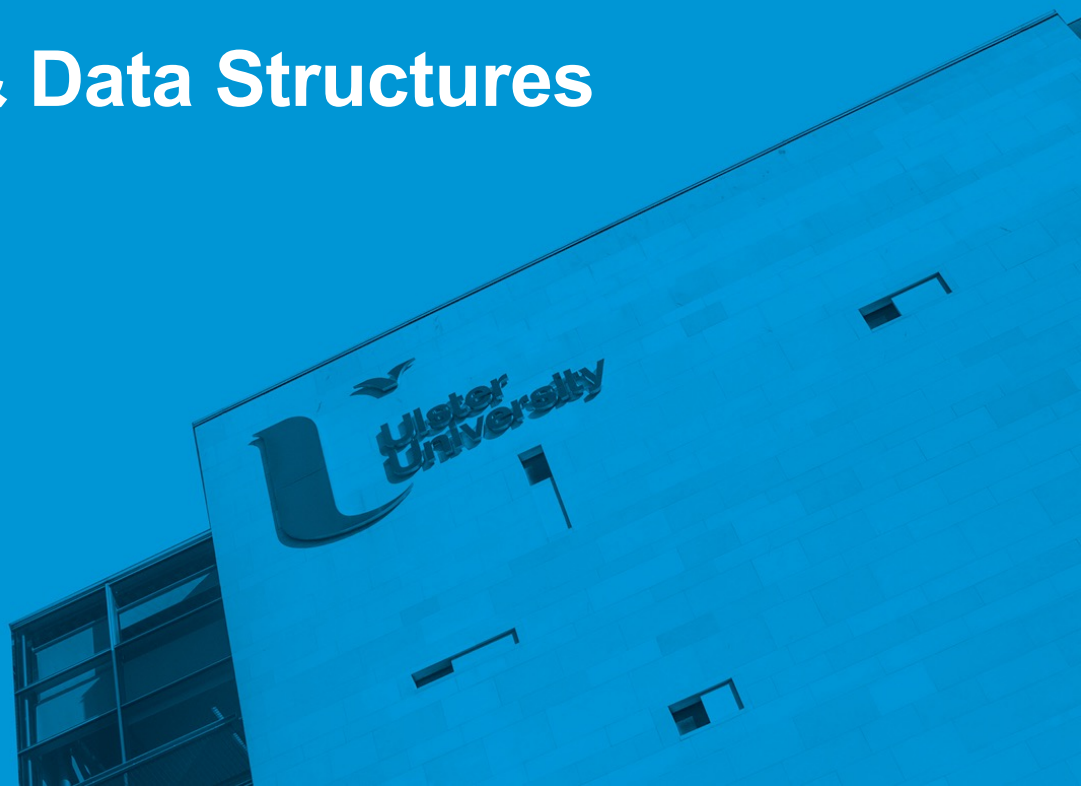
Ulster University

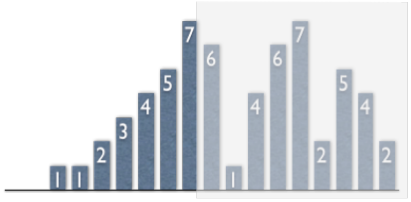
COM498 Algorithms & Data Structures

7.2 Shell Sort



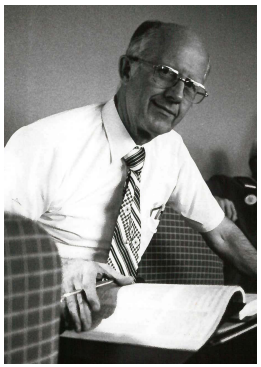
Shell Sort

- Algorithms seen so far are somewhat inefficient for large arrays



  - During an insertion sort, when an entry is far from its correct sorted position, it must make many adjacent moves – worst case scenario takes a good deal of time! ($O(n^2)$)

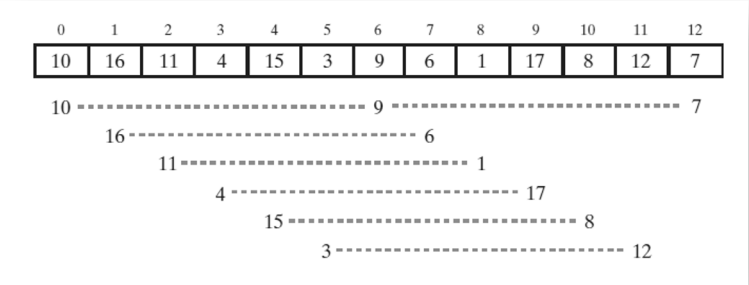
- When an array is almost sorted, insertion sort is much more efficient ($O(n)$)



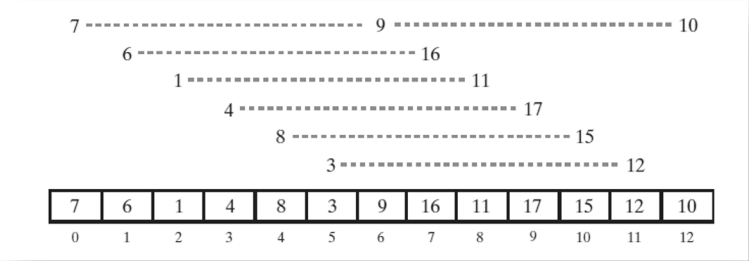
  - In 1959 Donald Shell devised an improved insertion sort (now known as Shell Sort) that permitted entries to move beyond their adjacent locations
  - To do this he sorted subarrays of entries at equally spaced indices (instead of moving to an adjacent location, an entry moved several locations away)

Shell Sort

- For example, an array of integers and the subarrays formed by grouping entries whose indices are 6 apart:



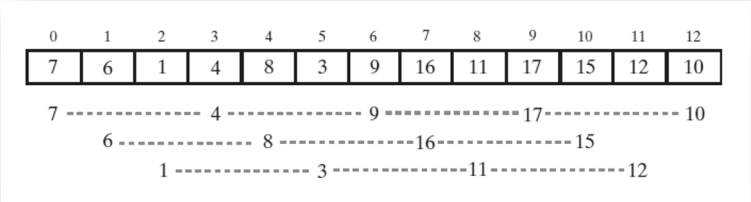
- First subarray contains integers: 10, 9, 7
- Second subarray contains integers: 16, 6, etc.
- There happen to be 6 subarrays, which are separately sorted using an Insertion Sort



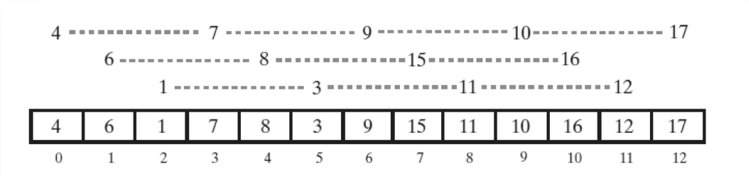
- The sorted subarrays and state of the original array as a result
- Array is more sorted than it was originally

Shell Sort

- Now we form new subarrays but reduce the separation between indices (n/2)
- For example, new subarrays formed by grouping entries whose indices are 3 apart:



- First new subarray contains integers 7, 4, 9, 17, 10, second subarray contains integers 6, 8, 16, 15, etc.
- Each time reduce separation by n/2 until it is 1



- The sorted subarrays and state of the original array as a result
- Final step (separation = 1) is simply an ordinary insertion sort of the entire array

Shell Sort

- The process of moving the entries by more than one position is also know as *h-sorting*
- Shell: *h-sort* array for decreasing sequence of values of *h,* where *h-sort* is an Insertion Sort with stride length *h*
- Further examples of sorting subarrays (h-sort) according to a separation value (h)

input	S	H	E	L	L	S	O	R	T	E	X	A	M	P	L	E
13-sort	P	H	E	L	L	S	O	R	T	E	X	A	M	S	L	E
4-sort	L	E	E	A	M	H	L	E	P	S	O	L	T	S	X	R
1-sort	A	E	E	E	H	L	L	L	M	O	P	R	S	S	T	X

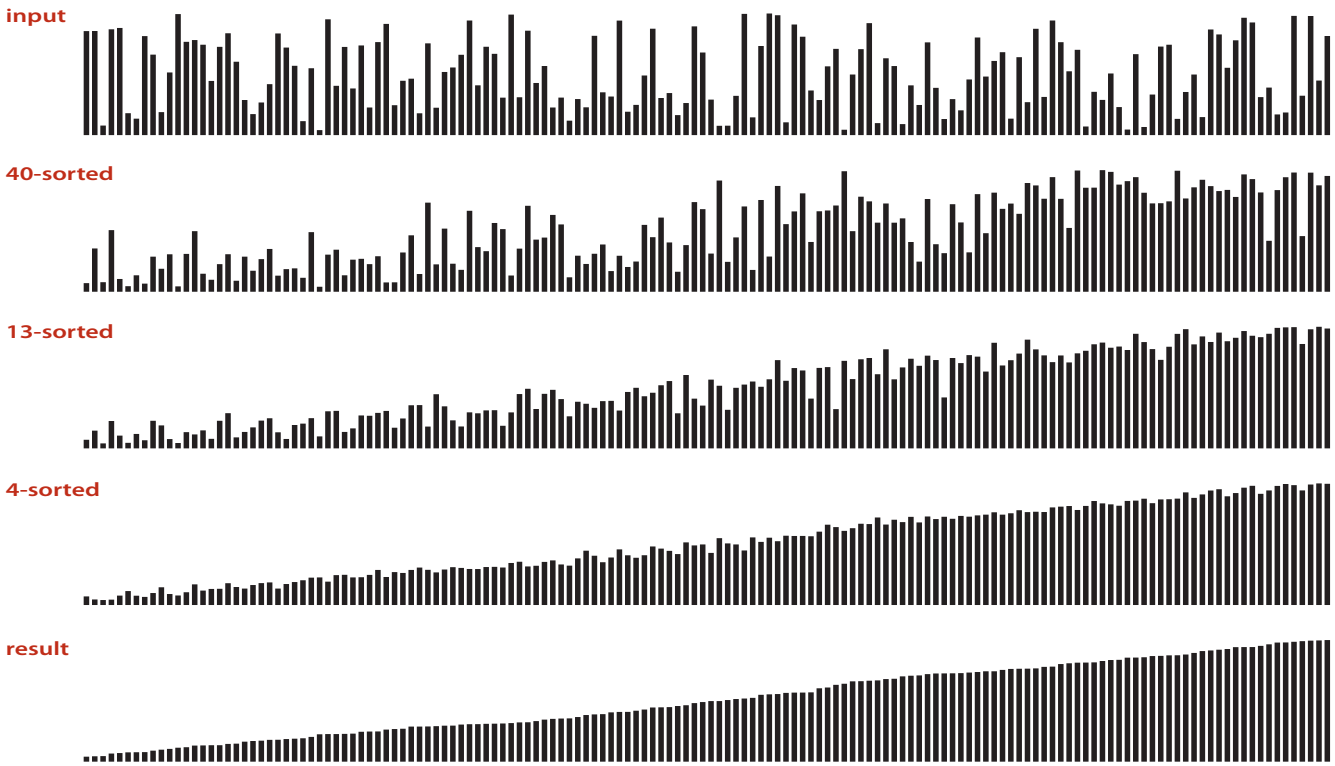
Shell Sort

- The process of moving the entries by more than one position is also know as *h-sorting*
- Shell: *h-sort* array for decreasing sequence of values of *h,* where *h-sort* is an Insertion Sort with stride length *h*
- Further examples of sorting subarrays (h-sort) according to a separation value (h)

input	S	H	E	L	L	S	O	R	T	E	X	A	M	P	L	E
13-sort	P	H	E	L	L	S	O	R	T	E	X	A	M	S	L	E
4-sort	L	E	E	A	M	H	L	E	P	S	O	L	T	S	X	R
1-sort	A	E	E	E	H	L	L	L	M	O	P	R	S	S	T	X

Shell Sort

- Visualization of the Shell Sort for various separations:



Shell Sort

Ulster University

- Pseudocode of algorithm for Shell Sort:

```
Algorithm shellSort(a)
// Sorts the entries of an array a.

loop gap from size of a / 2 to 1, dividing by 2

    // process groups spaced by gap
    loop i from gap to array size - 1
        set temp to a[i] // the next value to be inserted

        // find the place to insert the value
        loop index from i to gap while a[index - gap] > temp, by subtracting gap
            set a[index] to a[index - gap]
        set a[index] to temp
```

Trace of Shell Sort algorithm

9	2	7	1	10	3	6	4	5	8	**Gap = 5**
9	2	7	1	10	3	6	4	5	8	**Gap = 5**
3	2	7	1	10	9	6	4	5	8	**Gap = 5**
3	2	7	1	10	9	6	4	5	8	**Gap = 5**
3	2	4	1	10	9	6	7	5	8	**Gap = 5**
3	2	4	1	10	9	6	7	5	8	**Gap = 5**
3	2	4	1	8	9	6	7	5	10	**Gap = 5**

Trace of Shell Sort algorithm

3	2	4	1	8	9	6	7	5	10	**Gap = 2**
3	2	4	1	8	9	6	7	5	10	**Gap = 2**
3	2	4	1	8	9	6	7	5	10	**Gap = 2**
3	1	4	2	8	9	6	7	5	10	**Gap = 2**
3	1	4	2	8	9	6	7	5	10	**Gap = 2**
3	1	4	2	8	9	6	7	5	10	**Gap = 2**
3	1	4	2	6	9	8	7	5	10	**Gap = 2**
3	1	4	2	6	7	8	9	5	10	**Gap = 2**
3	1	4	2	5	7	6	9	8	10	**Gap = 2**
3	1	4	2	5	7	6	9	8	10	**Gap = 2**

Trace of Shell Sort algorithm

3	1	4	2	5	7	6	9	8	10	**Gap = 1**
3	1	4	2	5	7	6	9	8	10	**Gap = 1**
1	3	4	2	5	7	6	9	8	10	**Gap = 1**
1	3	4	2	5	7	6	9	8	10	**Gap = 1**
1	2	3	4	5	7	6	9	8	10	**Gap = 1**
1	2	3	4	5	7	6	9	8	10	**Gap = 1**
1	2	3	4	5	7	6	9	8	10	**Gap = 1**
1	2	3	4	5	6	7	9	8	10	**Gap = 1**
1	2	3	4	5	6	7	9	8	10	**Gap = 1**
1	2	3	4	5	6	7	8	9	10	**Gap = 1**
1	2	3	4	5	6	7	8	9	10	**Gap = 1**

Scenario

- Implement the Shell Sort
    - As for the previous sorts, add code to display the state of the array at every stage and to display the number of comparisons and array updates (changes to individual array elements).
    - Test the implementation for an integer array of 10 values where…
        1. the array to be sorted is in random order
        2. the array is already in sorted order
        3. the initial array is in reverse order

Efficiency of Shell Sort

Ulster University

- Since Shell Sort repeatedly uses Insertion Sort, would seem like much more work than using only one insertion sort, **BUT…**
    - Initial sorts are of arrays that are much smaller than original array (therefore quicker)
    - Later sorts are on arrays that are partially sorted, the final sort is on an almost entirely sorted array (therefore quicker)

- Calculating the time signature for Shell Sort is a very complex process which depends on the gap sequence used and the initial state of the array
    - Studies have shown a best case performance of **$O(n \log_2 n)$**
    - The worst case and average performances have been calculated as **$O(n (\log_2 n)^2)$**. This is often reported as **$O(n^2)$**.

Challenge

1. Measure the performance of the Shell Sort technique
   - Revisit the previous challenge and measure the execution time to sort 1000 arrays of 100, 200, 400, 800, 1600, 3200 and 6400 elements.
   - Report the results as a series of 2 values per line – array size and sort time.

2. The selection of gap sequences for Shell Sort is topic of hot debate and many alternatives have been proposed.
   - Compare some of the most popular gap sequences by measuring the time to sort 1000 arrays of 10000 elements, using the following sequences…
     a) n/2, n/4, n/8, …, 1 (as in our algorithm)
     b) n/3, n/6, n/9, … 1 (divide by 3 rather than by 2 at each stage)
     c) n/4, n/16, n/64, … 1 (divide by 4 at each stage)
     d) 3785, 1695, 749, 326, 138, 57, 23, 9, 4, 1 (mean of prime numbers sequence)
   - Show the execution time for each gap sequence