# Lecture 16
## Shell Sort

*October 18, 2021*
*Monday*

# MOTIVATION

- The ***O*** $(n^2)$ limit for sorting ***n*** elements is much too large.
- The time required for ordering an array, grows larger than the size of the array.
- We want to do Better !

# MOTIVATION

- It can be effective to sort parts of the original array first.
- And then, if they are partially sorted, sort the entire array.
- This way we are much closer to the best case scenario.

# PSEUDOCODE

```
divide data into h subarrays;
for i = 1 to h
        sort subarray data;
sort array data;
```

# SUBARRAYS

- If h is too small, then the subarrays $data_i$ of data could be too large.
  - The algorithm may prove inefficient.
- If h is too large, then too many small subarrays $data_i$ are created.
  - Although the subarrays are sorted.
  - It might not substantially change the overall order of data.
- Many techniques have been presented to find the best partition scheme.
  - The heart of **Shell Sort** is this ingenious division of the array into several subarrays.
  - Name after **Donald L. Shell** who designed this technique

# PSEUDOCODE

*determine numbers* $h_t$ … $h_1$ of ways of dividing array data into subarrays;

for (h = $h_t$ ; t > 1; t--, h = $h_t$ )

    *divide* data *into* h *subarrays;*

    for i = 1 to h

        sort subarray data;

*sort array* data;

# SUBARRAYS

- The original array is divided into subarrays logically
  - Every $h_t$ th element as part of one subarray.
- Therefore, array is divided into $h_t$ subarrays for every h = 1, …, $h_t$
  - $data_{h\_t}[\ i\ ] = data[\ h_t \times i + (\ h - 1)\ ]$
  - $data_{31}[\ 0\ ]$ = data [ 0 ], data [ 3 ], …
  - $data_{31}[\ i\ ]$ = data [ 3 * i + 1 - 1 ]

# SUBARRAYS PICTORIAL REPRESENTATION

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| `data` before 5-sort | 10 | 8 | 6 | 20 | 4 | 3 | 22 | 1 | 0 | 15 | 16 |
| Five subarrays before sorting | 10 | — | — | — | — | 3 | — | — | — | — | 16 |
| | | 8 | — | — | — | — | 22 | | | | |
| | | | 6 | — | — | — | — | 1 | | | |
| | | | | 20 | — | — | — | — | 0 | | |
| | | | | | 4 | — | — | — | — | 15 | |
| Five subarrays after sorting | 3 | — | — | — | — | 10 | — | — | — | — | 16 |
| | | 8 | — | — | — | — | 22 | | | | |
| | | | 1 | — | — | — | — | 6 | | | |
| | | | | 0 | — | — | — | — | 20 | | |
| | | | | | 4 | — | — | — | — | 15 | |

# SUBARRAYS PICTORIAL REPRESENTATION

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| `data` after 5-sort and before 3-sort | 3 | 8 | 1 | 0 | 4 | 10 | 22 | 6 | 20 | 15 | 16 |
| Three subarrays before sorting | 3 | — | — | 0 | — | — | 22 | — | — | 15 | |
| | | 8 | — | — | 4 | — | — | 6 | — | — | 16 |
| | | | 1 | — | — | 10 | — | — | 20 | | |
| Three subarrays after sorting | 0 | — | — | 3 | — | — | 15 | — | — | 22 | |
| | | 4 | — | — | 6 | — | — | 8 | — | — | 16 |
| | | | 1 | — | — | 10 | — | — | 20 | | |

# SUBARRAYS PICTORIAL REPRESENTATION

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| data after 3-sort and before 1-sort | 0 | 4 | 1 | 3 | 6 | 10 | 15 | 8 | 20 | 22 | 16 |
| data after 1-sort | 0 | 1 | 3 | 4 | 6 | 8 | 10 | 15 | 16 | 20 | 22 |

# PARTITIONING

- No formal proof indicating which sequence of increments is optimal.
- Good idea to Choose:

$$h_1 = 1$$

$$h_{i+1} = 3h_i + 1$$

Stop for $h_t$ if $h_t \geq n$

For n = 10,000

1, 4, 13, 40, 121, 364, 1093, 3280

# PARTITIONING

```
void ShellSort(int data[ ], int n) {
        int i, j,


}
```

# FEATURES

- The sequence of Increments.
- A simple sorting algorithm applied in all passes except the last.
- A simple sorting algorithm applied only in the last pass, for 1-sort.

# TIME COMPLEXITY

- The number of comparisons are same in each case
  - Best Case Comparisons: n ( n - 1 ) / 2 = ***O*** (n log n)
  - Average Case Comparisons: = ***O*** (n log n)
  - Worst Case Comparisons: ***O*** ($n^2$)

- Space Complexity ***O (1)***.

# LIMITATIONS

- One problem which remains open is the optimal value of the increment.
- Donald Knuth have shown even with two increments $(16\ n / \pi)^{1/3}$ and 1.
  - Shell sort is more efficient than insertion sort
  - Because it takes $\boldsymbol{O(n^{5/3})}$ instead of $\boldsymbol{O(n^2)}$
- The efficiency can be further improved by using a larger number of increments.
- We have to just consider one situation
  - The items in the even and odd positions of the array do not interact until the last pass, when the increment equals 1.

# VISUALIZATION


a[0]
a[1]
a[2]
a[3]
a[4]
a[5]
a[6]
a[7]
a[8]
a[9]
0
2
5
4
3
7
6
9
8