# Shell Sort:

*Transcending the Limitations of Incremental Sorting*

# ShellSort

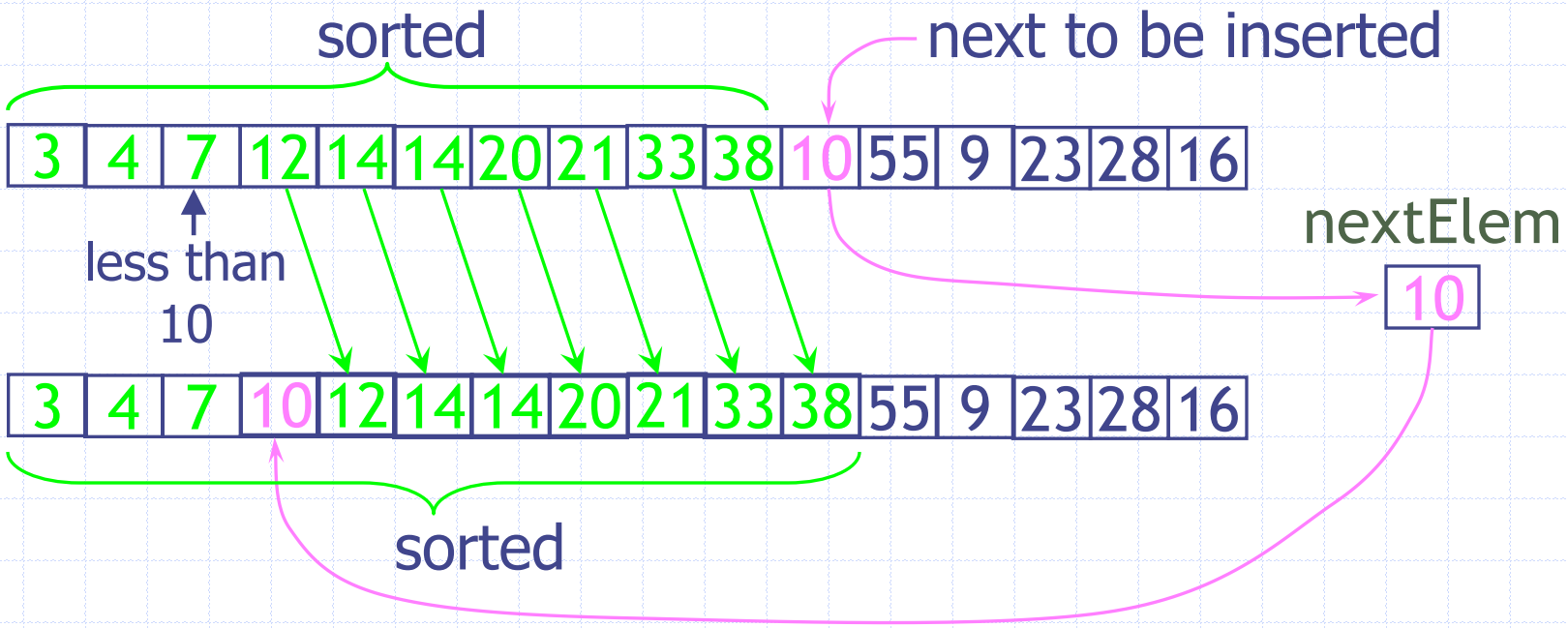
- Formulated by Donald Shell, who named the sorting algorithm after himself in 1959.
- A sorting algorithm based on InsertionSort.
- Much faster than the $O(n^2)$ sorts like SelectionSort and InsertionSort.
- Good for medium-sized arrays (as is InsertionSort generally), perhaps up to a few thousand items

# InsertionSort

**Algorithm** *InsertionSort*(*arr*)
**Input** Array *arr*
**Output** elements in *arr* are in sorted order

```
for i ← 1 to arr.length - 1 do
    j ← i
    nextElem ← arr[i]
    while 0 < j ∧ nextElem < arr[j - 1] do
        arr[j] ← arr[j - 1]        // shift element to right
        j ← j – 1
    arr[j] ← nextElem              // insert element into sorted location
```

# Inner While-Loop of InsertionSort




- This one step, the inner while-loop, could make O(i) shifts in the worst case

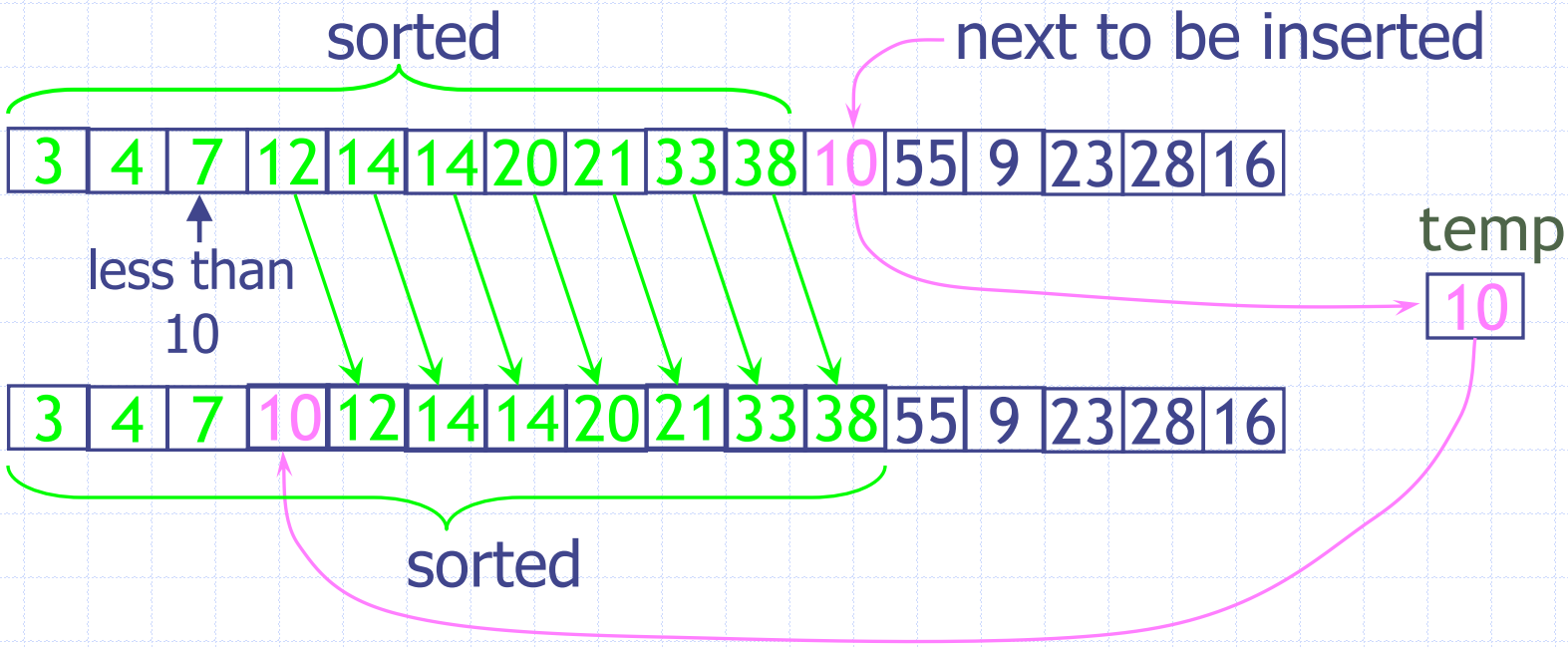
# Analysis of InsertionSort

- How many comparisons and shifts does this algorithm require? On the first pass, it compares a maximum of one item. On the second pass, it's a maximum of two items, and so on, up to a maximum of n-1 comparisons on the last pass. This is 1 + 2 + 3 + … + n-1 = n*(n-1)/2.
- The number of shifts is approximately the same as the number of comparisons. However, a shift/move operation isn't as expensive as a swap. In any case, like selection sort, the insertion sort runs in $\Theta(n^2)$ time for random data.
- However, because on each pass an average of only half of the maximum number of items are actually compared before the insertion point is found, we can divide by 2, which gives $n^2/4$ on average.

# Comparing Performance of Simple Sorting Algorithms

- Swaps are more expensive than shifts/moves. Notice that swaps involve roughly ten primitive operations. This is more costly than shifting (which takes about four).
- BubbleSort performs (on average) $O(n^2)$ swaps whereas SelectionSort performs only O(n) swaps, and InsertionSort does not perform any swaps at all (it shifts right which takes less than half as much time as a swap).
- Also, insertion sort does, on average, half as many key comparisons. Demos give empirical data for comparison.
- Empirical studies show that InsertionSort is 5 times faster than BubbleSort and 3.5 times faster that SelectionSort on average.

# Problem with InsertionSort: Too Many Shifts

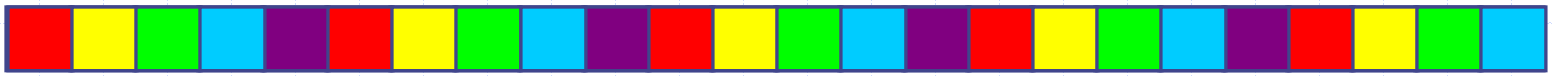



- This one step possibly makes more shifts than necessary.

# Problem with InsertionSort: Too Many Shifts (Could we do fewer?)

- Suppose a small item is on the far right.
- To move this small item to its proper place on the left, all the intervening items (between the place where it is and where it should be) must be compared and shifted one space to the right.
- If there are *k* intervening items, this step requires *k* shifts to handle one item. If *k* equals n, then it takes O(n) to move just one item to its proper place.
- This performance could be improved if we could somehow move a smaller item many spaces to the left without shifting all the intermediate items individually.

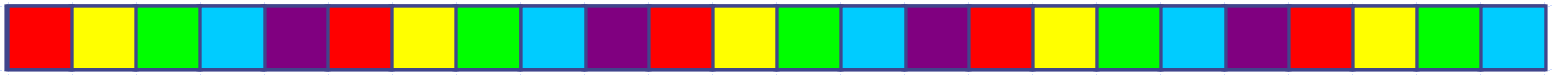
# ShellSort – General Description



- Essentially a segmented InsertionSort
  - Divides an array into several smaller noncontiguous segments
  - The distance between successive elements in one segment is called a *gap/Increment* (usually represented by h).
  - Each segment is sorted within itself using InsertionSort.
  - Then re-segment into larger segments (smaller gaps) and repeat sort.
  - Continue until there is only one segment (h= 1).

# Sorting nonconsecutive subarrays

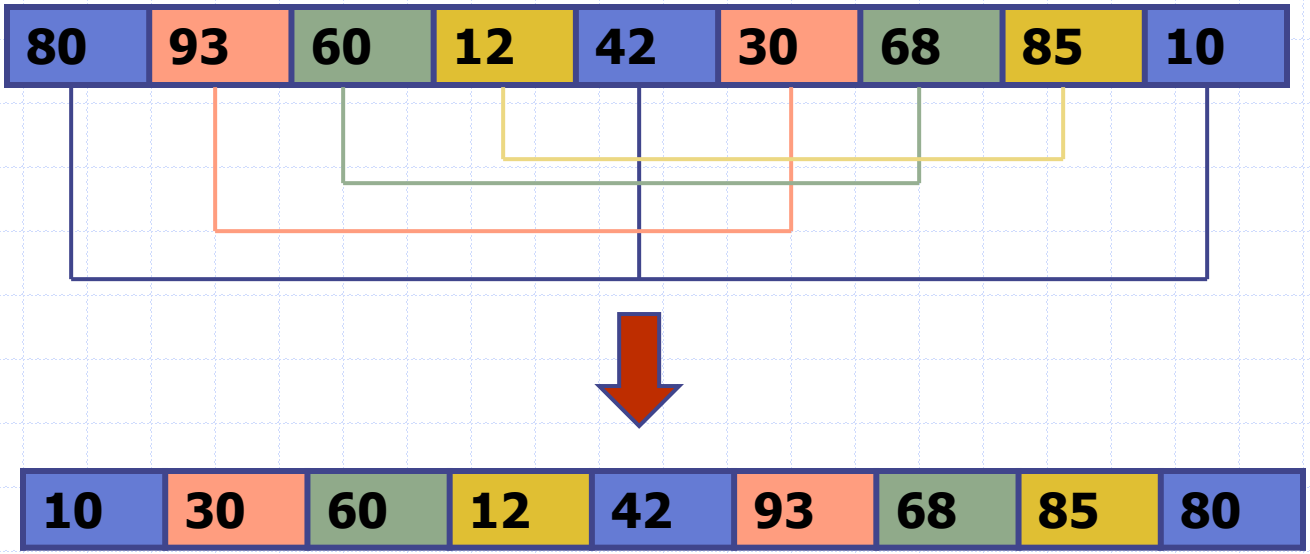
Here is an array to be sorted (numbers aren't important)



- Consider just the red locations.
- We try doing insertion sort on *just the red numbers,* as if they were the only ones in the array.
- Next do the same for just the yellow locations -- we do an insertion sort on just these numbers.
- Now do the same for each additional group of numbers.
- The resultant array is sorted *within groups,* but not overall.

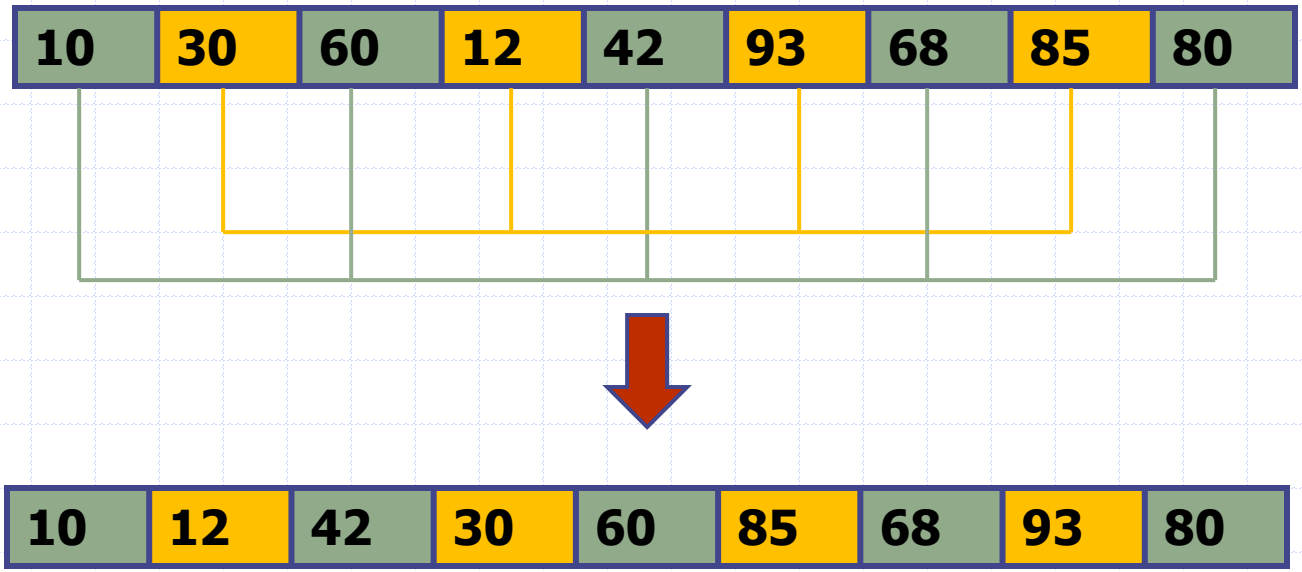
# Example

## Initial Segmenting Gap = 4



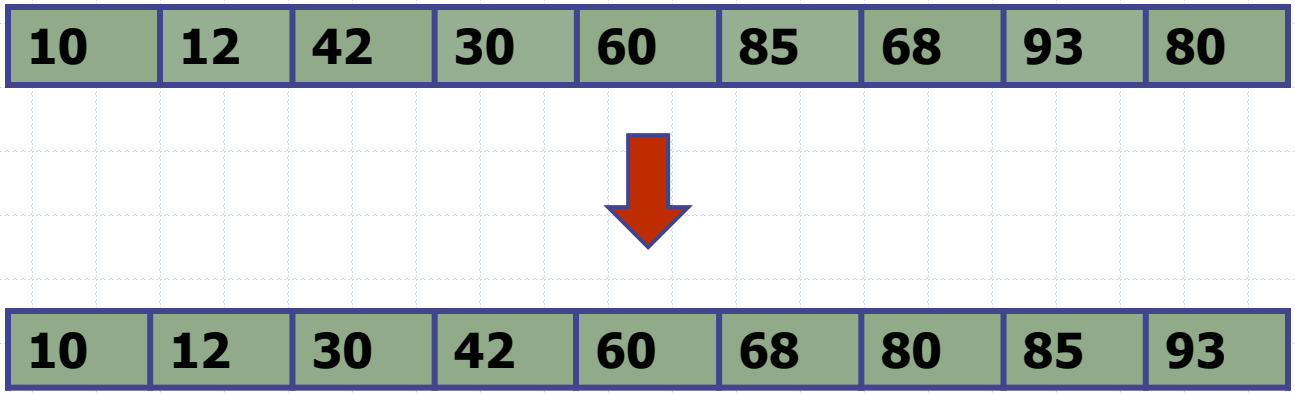
# Example

## Initial Segmenting Gap = 2



# Example

## Initial Segmenting Gap = 1



# Diminishing Gaps and N-Sorting

- In the example, we have seen an initial interval/gap of 4 cells for sorting a 9-cell array. This means that all items spaced four cells apart are sorted among themselves. We call it 4-sort. In general, applying InsertionSort on gap of N cells is called N-Sorting.
- The interval is then repeatedly reduced until it becomes 1.
- The set of intervals used in the example, (4, 2, 1) is called the *interval sequence* or *gap sequence*.
- Using this gap sequence, we can also say we did 4-sort, 2-sort, and 1-sort on the example.

# Obtaining a Gap Sequence

- Any decreasing gap sequence will work (if the last gap is 1), but the running time depends crucially on the choice of the gap sequence.

- When the gap sequence consists of powers of 2, such as (8, 4, 2, 1) (this was Shell's original method) it can be shown that the worst-case running time is no better than InsertionSort: $O(n^2)$. Running time is improved when terms of the gap sequence are relatively prime (no common factors other than 1).

# Added Gap Sequence

- Donald Knuth, in his discussion of Shell's Sort, recommended another sequence of gaps.

$$h_0 = 1,\ h_{j+1} = h_j * 3 + 1$$

  - Find the largest $h_j <= n$, then start with $h_j$

| h | 3*h + 1 | (h–1) / 3 |
|---|---|---|
| 1 | 4 | |
| 4 | 13 | 1 |
| 13 | 40 | 4 |
| 40 | 121 | 13 |
| 121 | 364 | 40 |
| 364 | 1093 | 121 |
| 1093 | 3280 | 364 |

For example, sorting a 1,000-element array, first needs to find a largest h <= n, which will be 364 in this case. Then, reduce the interval using the inverse of the formula given:

$$h = (h–1) / 3$$

This inverse formula generates the reverse sequence 364, 121, 40, 13, 4, 1.

# ShellSort Algorithm

```
Algorithm shellSort(arr)
   Input Array arr
   Output elements of arr are rearranged into sorted order
      maxGap ← floor((arr.length-1)/3)
      h ← 1
      while (h ≤ maxGap) do  // compute the maximum gap size
            h ← h * 3 + 1  // 1,4,13,40,121,…
      while (0 < h) do
            segmentInsertionSort(arr, h)
            h ← (h - 1) / 3  // …,121,40,13,4,1

Algorithm segmentInsertionSort(arr, gap)
    for  i ← gap  to arr.length - 1  do
        j ← i
        nextElem ← arr[i]
        while  (gap - 1) < j  ∧ nextElem < arr[j - gap]  do
                  arr[j] ← arr[j - gap]        // shift element to right
                  j ← j – gap
        arr[j] ← nextElem
```

# Running time

- Real running time of Shellsort? Although running times for ShellSort using certain gap sequences are known, finding the gap sequence that produces the best possible running time is still being researched.
- For any version of ShellSort, we do know that its average running time is $O(n^r)$ with $1 < r < 2$
- Generally speaking, ShellSort's running time is better than $O(n^2)$ but worse than O(nlog n).

# Ideal Gap Sequence

Although mathematical techniques have been developed to optimize the gap sequence used, the best gap sequences have been found just by empirical tests. Here are a few of the best known results [see https://en.wikipedia.org/wiki/Shellsort]

| Concrete gaps | Worst-case time complexity | Author and year of publication |
|---|---|---|
| $\left\lfloor \frac{N}{2} \right\rfloor, \left\lfloor \frac{N}{4} \right\rfloor, \ldots, 1$ | $\Theta(N^2)$ [when $N=2^p$] | Shell, 1959[3] |
| $2\left\lfloor \frac{N}{4} \right\rfloor + 1, \ldots, 3, 1$ | $\Theta(N^{3/2})$ | Frank & Lazarus, 1960[7] |
| $1, 3, 7, 15, 31, 63, \ldots$ | $\Theta(N^{3/2})$ | Hibbard, 1963[8] |
| $1, 3, 5, 9, 17, 33, 65, \ldots$ | $\Theta(N^{3/2})$ | Papernov & Stasevich, 1965[9] |
| $1, 2, 3, 4, 6, 8, 9, 12, \ldots$ | $\Theta(N \log^2 N)$ | Pratt, 1971[10] |
| $1, 4, 13, 40, 121, \ldots$ | $\Theta(N^{3/2})$ | Pratt, 1971[10] |

# Main Point

1. The first algorithm to overcome the limitations of the simple sorting algorithms – BubbleSort, SelectionSort, Insertion Sort – was ShellSort. The strategy used in ShellSort was to remove one of the known limitations of the InsertionSort algorithm. The technique results in a significant jump in performance. *Science of Consciousness*: This step in the history of sorting algorithms illustrates the general principle that removal of blockages to optimal functioning of a system can greatly improve its performance. This is the strategy used by TM, which results in significant improvements in performance in life: intelligence, efficiency, satisfaction.

## Connecting the Parts of Knowledge
## With the Wholeness of Knowledge

### Simple Sorting

1. Insertion Sort sorts by examining each successive value x in the input list and searches the already sorted section of the array for the proper location for x.
2. ShellSort is also a sorting algorithm that performs the same steps as InsertionSort, but, before carrying out those steps, performs a number of pre-processing steps, called *n-sorting.* The result of this refinement is that, even in the worst case, many versions of ShellSort run in $O(n^r)$ for $r < 2$ or $r < 1.5$.
3. *Transcendental Consciousness* is the silent field of pure intelligence, the basis of all activity, basis of the physical law of least action.
4. *Impulses within the Transcendental field*. Contact with transcendental consciousness enlivens the support of the laws of nature for activity in life. Therefore, the "pre-processing" step necessary for success in life is *transcend*, then activity will be smooth and more successful.
5. *Wholeness moving within itself*. In Unity Consciousness, the transcendental level has already been automatically integrated with ordinary active awareness – no "pre-processing" step is needed in this state.