# Data Structure

## Lecture#16: Internal Sorting (Chapter 7)

**U Kang**
**Seoul National University**

# In This Lecture

- Definition and evaluation measures of sorting
- Exchange sorting algorithms and their limitations
- Shellsort and how to exploit the best-case behavior of other algorithm

# Sorting

- Sorting: puts elements of a list in a certain order (increasing or decreasing)
  - Many applications: scores, documents, search results, …
  - One of the most fundamental tasks in Computer Science

- Sorting in offline world

# Sorting

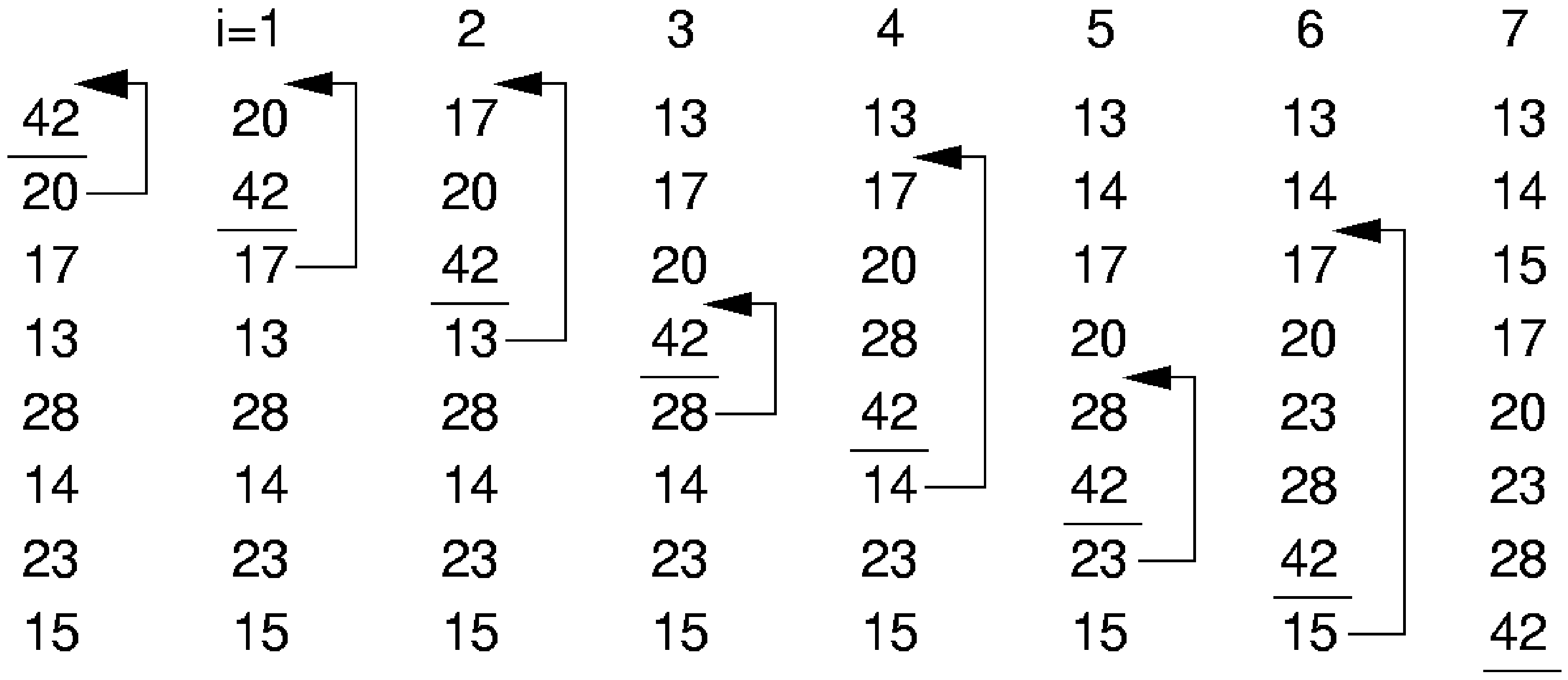
- We will discuss many sorting algorithms
    - insertion sort, bubble sort, selection sort, shell sort, merge sort, quicksort, heap sort, bin sort, radix sort

- Measures of cost:
    - # of Comparisons
    - # of Swaps

# Insertion Sort (1)

- Initially, the output is empty
- Insert each item one by one to the output
  - Insert it in a correct place to make the output in a sorted order

# Insertion Sort (2)



|  | i=1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 42 | 20 | 17 | 13 | 13 | 13 | 13 | 13 |
| 20 | 42 | 20 | 17 | 17 | 14 | 14 | 14 |
| 17 | 17 | 42 | 20 | 20 | 17 | 17 | 15 |
| 13 | 13 | 13 | 42 | 28 | 20 | 20 | 17 |
| 28 | 28 | 28 | 28 | 42 | 28 | 23 | 20 |
| 14 | 14 | 14 | 14 | 14 | 42 | 28 | 23 |
| 23 | 23 | 23 | 23 | 23 | 23 | 42 | 28 |
| 15 | 15 | 15 | 15 | 15 | 15 | 15 | 42 |

Input

Output

# Insertion Sort (3)

```
static <E extends Comparable<? super E>>
void Sort(E[] A) {
  for (int i=1; i<A.length; i++)
    for (int j=i;
         (j>0) && (A[j].compareTo(A[j-1])<0);
         j--)
      DSutil.swap(A, j, j-1);
}
```

# of Swaps, # of Comparisons

- Best Case:
- Worst Case:
- Average Case:

# Insertion Sort (4)

```
static <E extends Comparable<? super E>>
void Sort(E[] A) {
  for (int i=1; i<A.length; i++)
    for (int j=i;
         (j>0) && (A[j].compareTo(A[j-1])<0);
         j--)
      DSutil.swap(A, j, j-1);
}
```

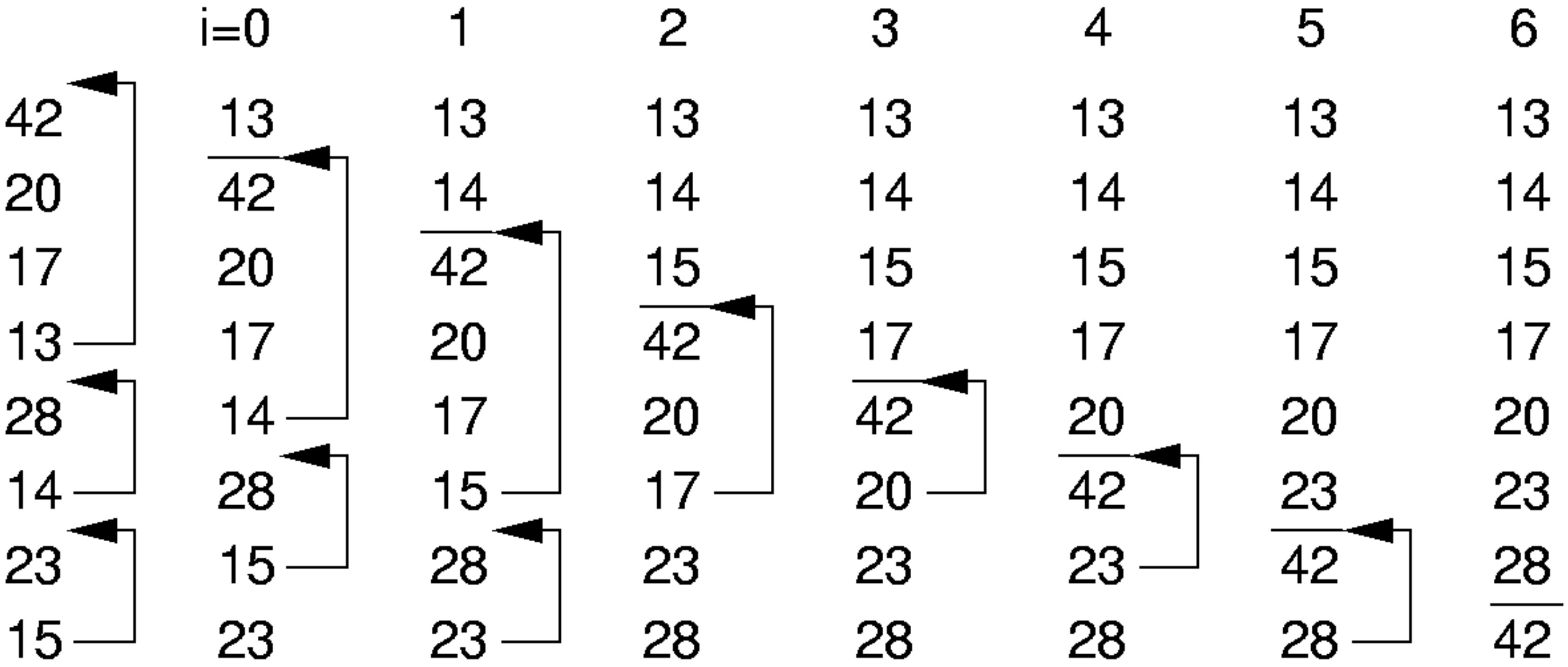
- Best Case: 0 swaps, n – 1 comparisons
- Worst Case: $n^2/2$ swaps and comparisons
- Average Case: $n^2/4$ swaps and comparisons

**Insertion Sort is very efficient when the array is near-sorted. This characteristic is used later in other sorting algorithms.**

# Bubble Sort (1)

- Maybe, one of the most popular sorting algorithms
    - Appears in many computer language introduction books
- Main Idea
    - Initially, the output is empty
    - At each iteration
        - "Bubble up" the smallest element from the input to the output (= move the smallest element from the input to the output)
    - Using an array for both input and output
        - At iteration *k*, *k* th smallest element is located in the array[k]
    - Given an array, how to move the smallest element to the beginning of the array?
        - One idea is to swap neighbors repeatedly, from the end of the array

# Bubble Sort (2)



| | i=0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| 42 | 13 | 13 | 13 | 13 | 13 | 13 | 13 |
| 20 | 42 | 14 | 14 | 14 | 14 | 14 | 14 |
| 17 | 20 | 42 | 15 | 15 | 15 | 15 | 15 |
| 13 | 17 | 20 | 42 | 17 | 17 | 17 | 17 |
| 28 | 14 | 17 | 20 | 42 | 20 | 20 | 20 |
| 14 | 28 | 15 | 17 | 20 | 42 | 23 | 23 |
| 23 | 15 | 28 | 23 | 23 | 23 | 42 | 28 |
| 15 | 23 | 23 | 28 | 28 | 28 | 28 | 42 |

# Bubble Sort (3)

```
static <E extends Comparable<? super E>>
void Sort(E[] A) {
  for (int i=0; i<A.length-1; i++)
    for (int j=A.length-1; j>i; j--)
      if ((A[j].compareTo(A[j-1]) < 0))
        DSutil.swap(A, j, j-1);
}
```

**# of Swaps, # of Comparisons**

- Best Case:
- Worst Case:
- Average Case:

# Bubble Sort (4)

```
static <E extends Comparable<? super E>>
void Sort(E[] A) {
  for (int i=0; i<A.length-1; i++)
    for (int j=A.length-1; j>i; j--)
      if ((A[j].compareTo(A[j-1]) < 0))
        DSutil.swap(A, j, j-1);
}
```

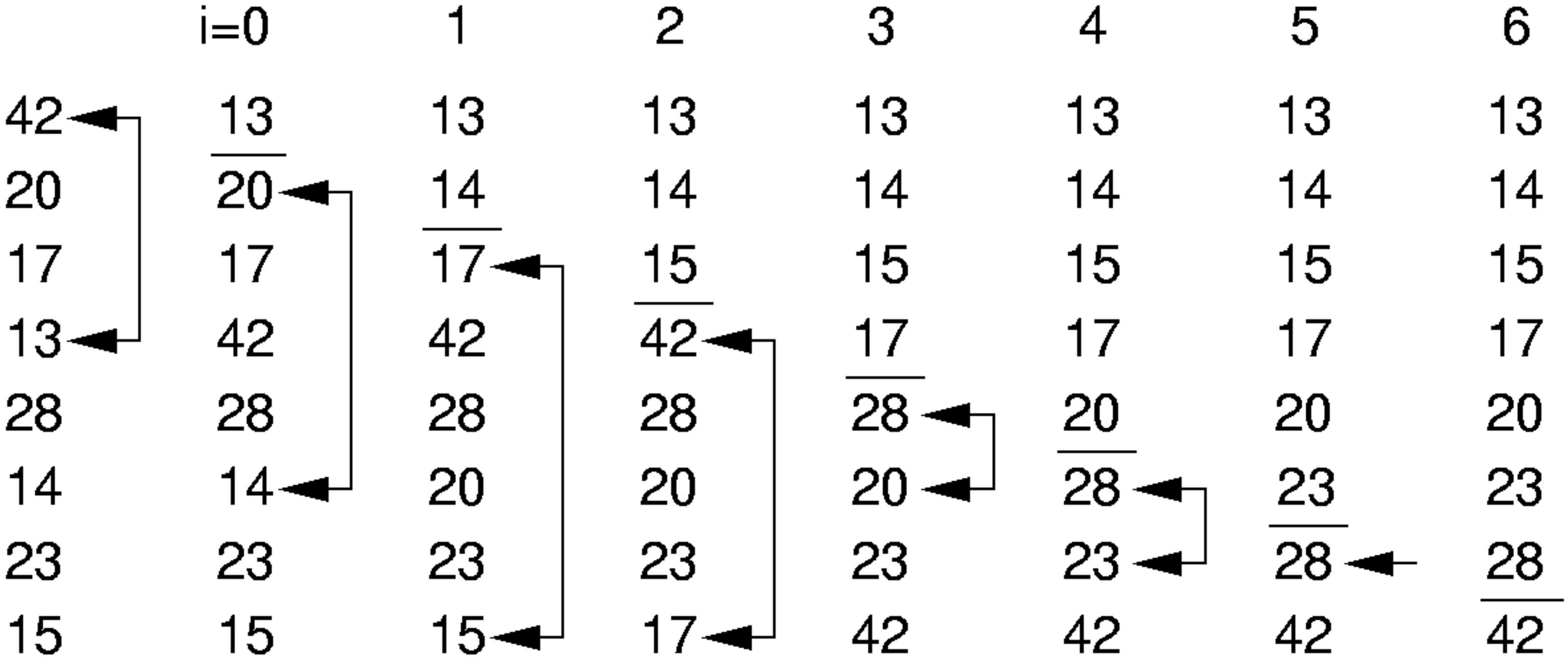
- Best Case: 0 swaps, $n^2/2$ comparisons
- Worst Case: $n^2/2$ swaps and comparisons
- Average Case: $n^2/4$ swaps and $n^2/2$ comparisons

# Selection Sort (1)

- Essentially, a bubble sort

- Given an array, how to move the smallest element to the beginning of the array?
  - [Bubble Sort] swap neighbors repeatedly
  - [Selection Sort] scan the array, find the smallest element, and swap it with the first item in the array

# Selection Sort (2)



| | i=0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| 42 | 13 | 13 | 13 | 13 | 13 | 13 | 13 |
| 20 | 20 | 14 | 14 | 14 | 14 | 14 | 14 |
| 17 | 17 | 17 | 15 | 15 | 15 | 15 | 15 |
| 13 | 42 | 42 | 42 | 17 | 17 | 17 | 17 |
| 28 | 28 | 28 | 28 | 28 | 20 | 20 | 20 |
| 14 | 14 | 20 | 20 | 20 | 28 | 23 | 23 |
| 23 | 23 | 23 | 23 | 23 | 23 | 28 | 28 |
| 15 | 15 | 15 | 17 | 42 | 42 | 42 | 42 |

# Selection Sort (3)

```
static <E extends Comparable<? super E>>
void Sort(E[] A) {
  for (int i=0; i<A.length-1; i++) {
    int lowindex = i;
    for (int j=A.length-1; j>i; j--)
      if (A[j].compareTo(A[lowindex]) < 0)
        lowindex = j;
    DSutil.swap(A, i, lowindex);
  }
}
```

**# of Swaps, # of Comparisons**

- Best Case:
- Worst Case:
- Average Case:

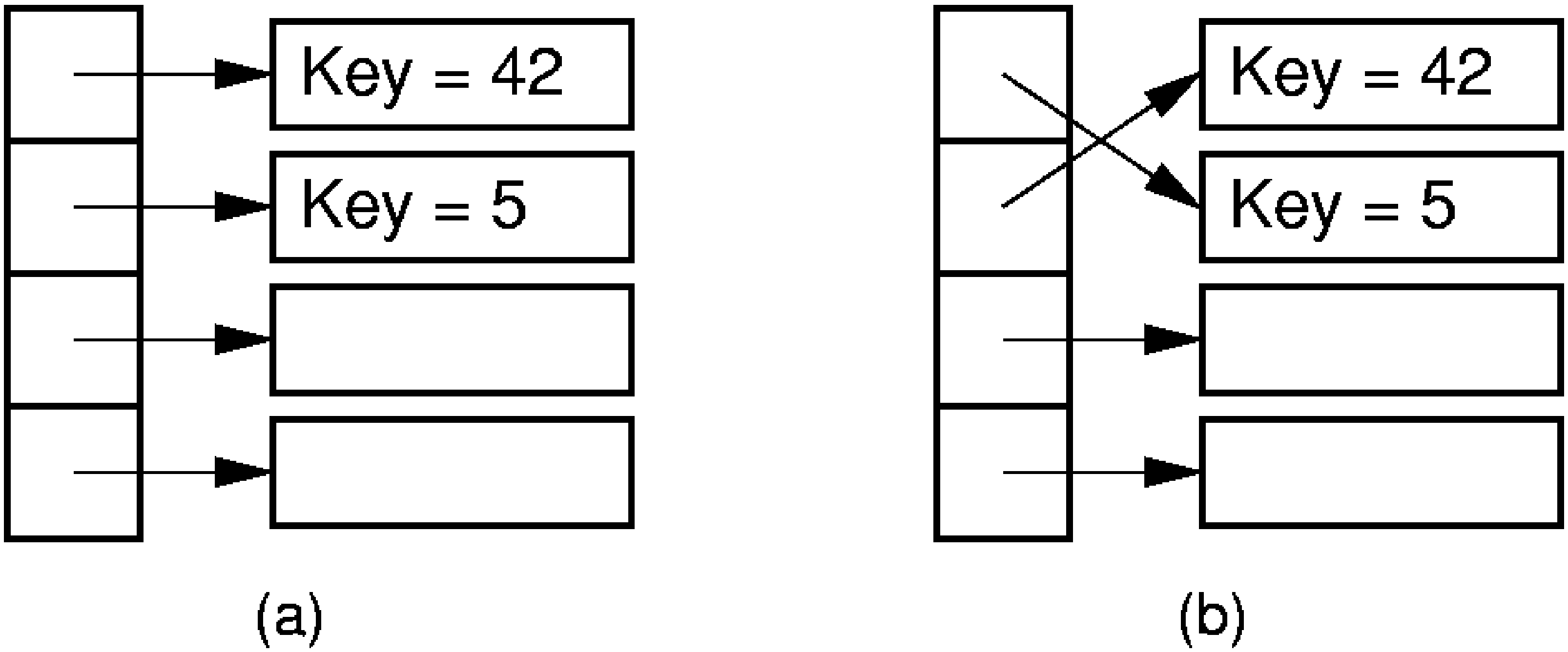
# Selection Sort (4)

```
static <E extends Comparable<? super E>>
void Sort(E[] A) {
  for (int i=0; i<A.length-1; i++) {
    int lowindex = i;
    for (int j=A.length-1; j>i; j--)
      if (A[j].compareTo(A[lowindex]) < 0)
        lowindex = j;
    DSutil.swap(A, i, lowindex);
  }
}
```

- Best Case: 0 swaps (n-1 swaps for bad swap()), $n^2/2$ comparisons
- Worst Case: n-1 swaps and $n^2/2$ comparisons
- Average Case: O(n) swaps and $n^2/2$ comparisons

**Better than Bubble sort, since # of swap is much smaller**

# Pointer Swapping



# Summary

| | Insertion | Bubble | Selection |
|---|---|---|---|
| **Comparisons** | | | |
| Best Case | $\Theta(n)$ | $\Theta(n^2)$ | $\Theta(n^2)$ |
| Average Case | $\Theta(n^2)$ | $\Theta(n^2)$ | $\Theta(n^2)$ |
| Worst Case | $\Theta(n^2)$ | $\Theta(n^2)$ | $\Theta(n^2)$ |
| **Swaps** | | | |
| Best Case | 0 | 0 | 0 or $\Theta(n)$ |
| Average Case | $\Theta(n^2)$ | $\Theta(n^2)$ | $\Theta(n)$ |
| Worst Case | $\Theta(n^2)$ | $\Theta(n^2)$ | $\Theta(n)$ |

# Exchange Sorting

- All of the sorting algorithms so far rely on exchanges of *adjacent* records.
  - Thus, they are called “exchange sorting” algorithms

- What is the average number of exchanges required in any exchange sorting of *n* items?
  - There are *n*! permutations
  - Consider a permuation *X* and its reverse, *X*’
  - Together, all pairs require *n*(*n*-1)/2 exchanges (or “inversion”) in total.
  - On average, each permutation requires *n*(*n*-1)/4 = $\Omega(n^2)$ exchanges

# Shell Sort (1)

- Main idea
  - Task: sort an array x of size n
  - Consider the following two sub arrays from x
    - $x_e$ (contains elements whose indexes are even)
    - $x_o$ (contains elements whose indexes are odd)
  - Assume $x_e$ and $x_o$ are sorted, respectively
  - Then, insertion sort on x would be efficient (why?)
  - Now, recursively consider the above process on the two subarrays

  - Shell sort: go backward from the end of the above process

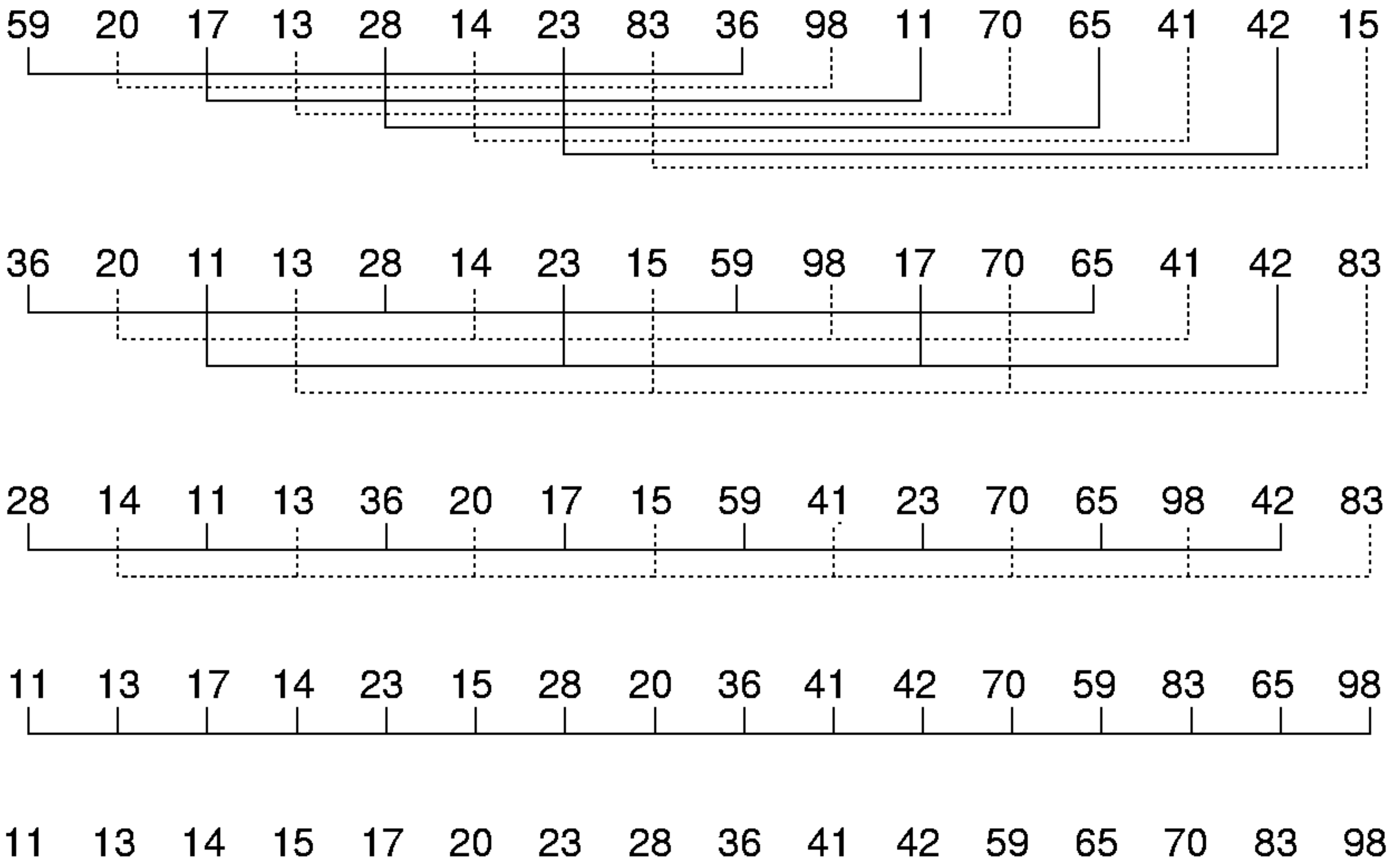
# Shell Sort (2)

- Procedure
  - Pass 1
    - Make n/2 sublists of 2 elements each, where the array index of the 2 elements differs by n/2
      - E.g., for n = 16, make 8 sublists: (0, 8), (1, 9), …, (7, 15)
    - Each list of 2 elements is sorted using Insertion Sort
  - Pass 2
    - Make n/4 sublists of 4 elements each, where the array index of the 4 elements differs by n/4
      - E.g., for n = 16, make 4 sublists: (0, 4, 8, 12), (1, 5, 9, 13), …
    - Each list of 4 elements is sorted using Insertion Sort
  - …

# Shell Sort (3)

- Main Idea
  - Pass 3
    - Make n/8 sublists of 8 elements each, where the array index of the 8 elements differs by n/8
      - E.g., for n = 16, make 2 sublists: (even numbers), (odd numbers)
    - Each list of 8 elements is sorted using Insertion Sort
  - … Final Pass (Pass (log *n*))
    - Make 1 sublist of n elements(=do nothing), and sort the sublist using insertion sort ( = apply the standard insertion sort on the array)

# Shell Sort (4)



59 20 17 13 28 14 23 83 36 98 11 70 65 41 42 15

36 20 11 13 28 14 23 15 59 98 17 70 65 41 42 83

28 14 11 13 36 20 17 15 59 41 23 70 65 98 42 83

11 13 17 14 23 15 28 20 36 41 42 70 59 83 65 98

11 13 14 15 17 20 23 28 36 41 42 59 65 70 83 98

# Shell Sort (5)

```
static <E extends Comparable<? super E>>
void Sort(E[] A) {
  for (int i=A.length/2; i>2; i/=2)
    for (int j=0; j<i; j++)
      inssort2(A, j, i);
  inssort2(A, 0, 1);
}

/** Modified version of Insertion Sort for
    varying increments */
static <E extends Comparable<? super E>>
void inssort2(E[] A, int start, int incr) {
  for (int i=start+incr; i<A.length; i+=incr)
    for (int j=i;(j >= start+incr)&&
                 (A[j].compareTo(A[j-incr])<0);
         j-=incr)
      DSutil.swap(A, j, j-incr);
}
```

# Shell Sort (6)

- Correctness: Shellsort always sorts an array correctly. Why?
  - Since it performs the insertion sort at the end

- Efficiency: Is Shellsort better than Insertion Sort?
  - Yes (in most cases), since each insertion sort operates on an "almost sorted" array
  - Fact: average-case performance of ShellSort takes $O(n^{1.5})$, which is much efficient than Insertion Sort

# What you need to know

- Sorting: puts elements in a certain order
    - Evaluation: # of swaps, # of comparisons
- Exchange sorting algorithms
    - Insertion sort, bubble sort, and selection sort
    - Cost and limitations
- Shellsort
    - Main ideas
    - How it exploits insertion sort

# Questions?