SAN FRANCISCO STATE UNIVERSITY
COMPUTER SCIENCE DEPARTMENT

# SORTING: SELECTION, INSERTION, SHELL

DUC TA

EXPERIENTIA DOCET

# Generic Types?

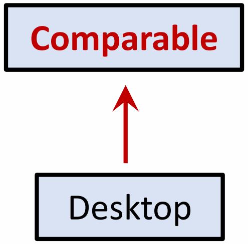
```
// Selection Sort
public static <T extends Comparable<? super T>> void selectionSort(T[] a, int n){
```

## Bounded Type Parameters

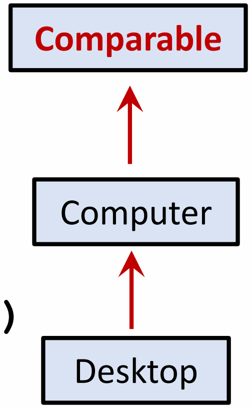
```
// Selection Sort, too restrictive

public static <T extends Comparable<T>> T selectionSort(T[] a, int n){
```

- Objects in the passed array be strings, Integer Objects, or any **comparable** objects.
- The class of these objects must implement the interface **Comparable**.
- **T** is bounded to represent class type that provides the method **compareTo()**.




- The green (1st) **T** is the type parameter's name
- The blue (2nd) **T** is the upper bound of the green (1st) T
- The brown (3rd) **T** is the return type (if used or if not void)
- The black (4th) **T** is the parameter type




- The **`extends`** keyword used in general sense to mean either extends (as in classes) or implements (as in interfaces). "implements" in this case.

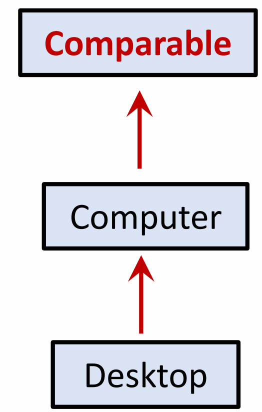
- Problems: it is **more restrictive** than necessary
  - **`compareTo`** only works for T, but not subclasses or super class of T.
  - **`Desktop smallestDesktop = MyClass.arrayMinimum(myArray)`**
  - Insisting comparing a Desktop vs. a Desktop, not even a Computer

## Bounded WildCard

```
    // CORRECT
    public static <T extends Comparable<? super T>> T arrayMinimum(T[] anArray) {
```

- Problems: it is **more restrictive** than necessary
  - `Desktop smallestDesktop = MyClass.arrayMinimum(myArray)`

- `<? super T>` allows comparing Desktop vs. Computer. That is Desktop vs. an object of Desktop's superclass.
- `<? super T>` means **any superclass** of T
- `<? super T>` T is the **lower bound** of the wild card "`?`"
- `? extends T` T is the **upper bound** of the wild card "`?`"




```
// Selection Sort
public static <T extends Comparable<? super T>> void selectionSort(T[] a, int n){
```

- T[] a: an array of Comparable objects
- int n: the first n objects in the array

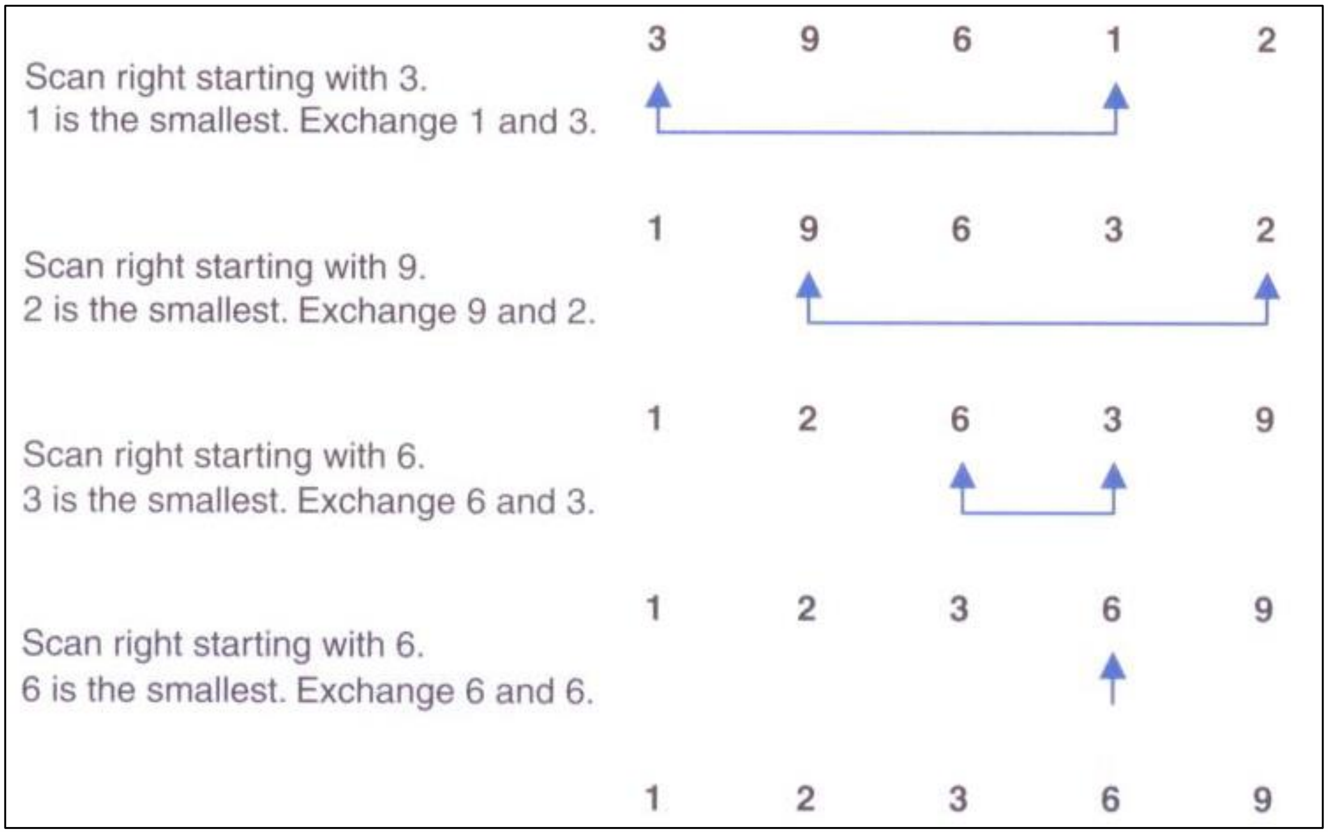
# SELECTION SORT


Selection Sort

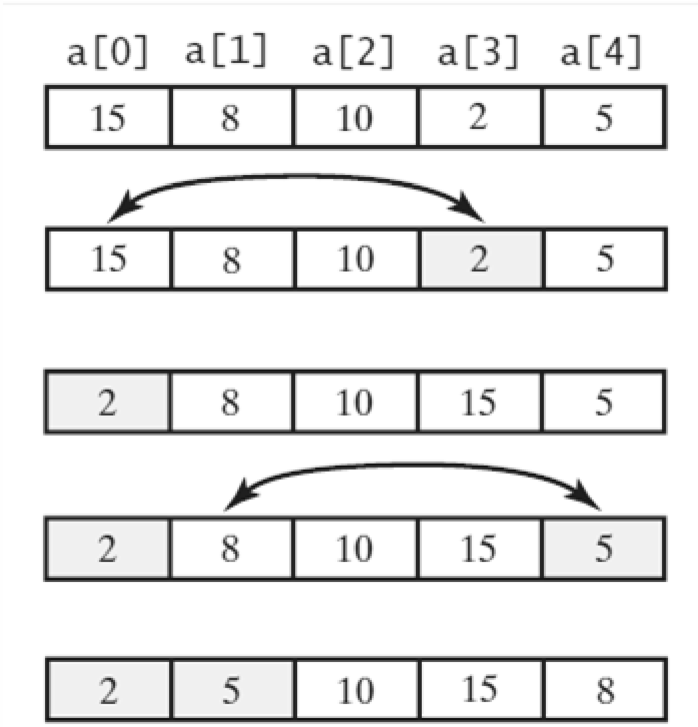
- **Selection sort** orders a list of values by repetitively **putting a particular value into its final position**
- More specifically:
    1. **Find** the **smallest** value in **the list**
    2. **Switch** it with the value in **the first position**
    3. **Find** the next **smallest** value in **the list**
    4. **Switch** it with the value in **the second position**
    5. Repeat **until all values are in their proper places**

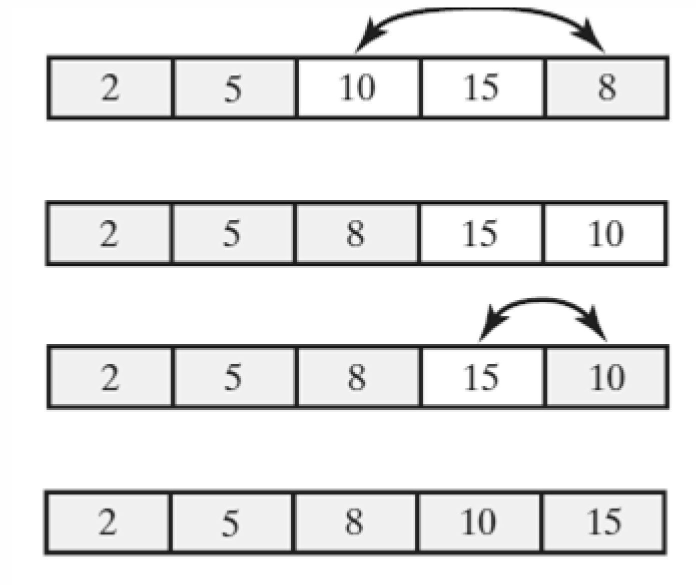
**Smallest → Final Position**

1. **Find** the **smallest** value in **the list**
2. **Switch** it with the value in **the first position**
3. **Find** the next **smallest** value in **the list**
4. **Switch** it with the value in **the second position**
5. Repeat **until all values are in their proper places**



# Selection Sort



```java

    // Finds the index of the smallest value in a portion of an array a.
    // Returns the index of the smallest value among

    private static <T extends Comparable<? super T>>
            int getIndexOfSmallest(T[] a, int first, int last) {

        T min = a[first];
        int indexOfMin = first;

        for (int index = first + 1; index <= last; index++) {

            if (a[index].compareTo(min) < 0) {
                min = a[index];
                indexOfMin = index;
            }
        }

        return indexOfMin;
    }


```

```java

public static <T extends Comparable<? super T>> void selectionSort(T[] a, int n) {

        for (int index = 0; index < n - 1; index++) {

            int indexOfNextSmallest = getIndexOfSmallest(a, index, n - 1);

            swap(a, index, indexOfNextSmallest);
        }
}

```

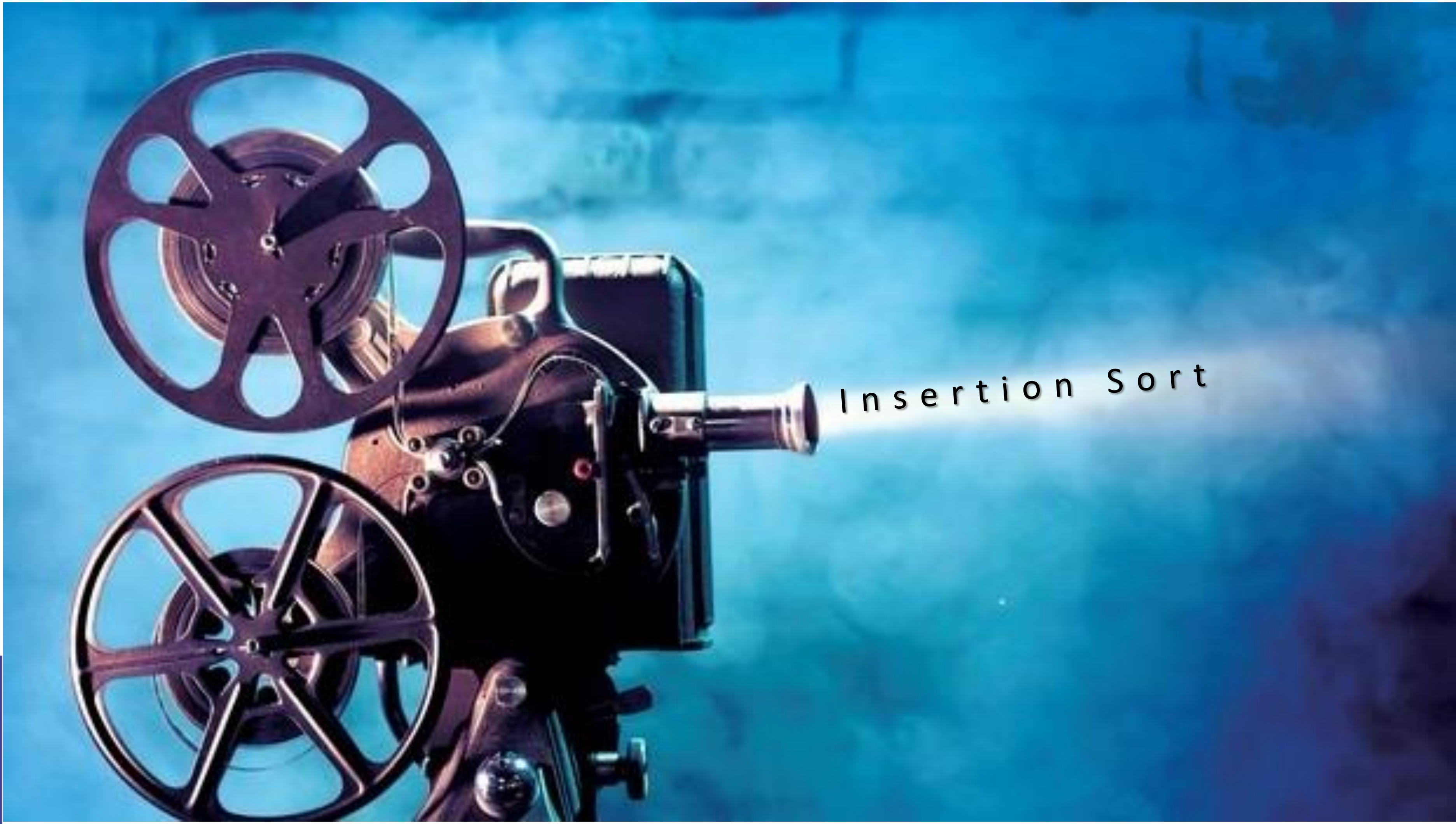
```java

    // Swaps the array entries a[i] and a[j].
    private static void swap(Object[] a, int i, int j) {
        Object temp = a[i];
        a[i] = a[j];
        a[j] = temp;
    }

```

# INSERTION SORT

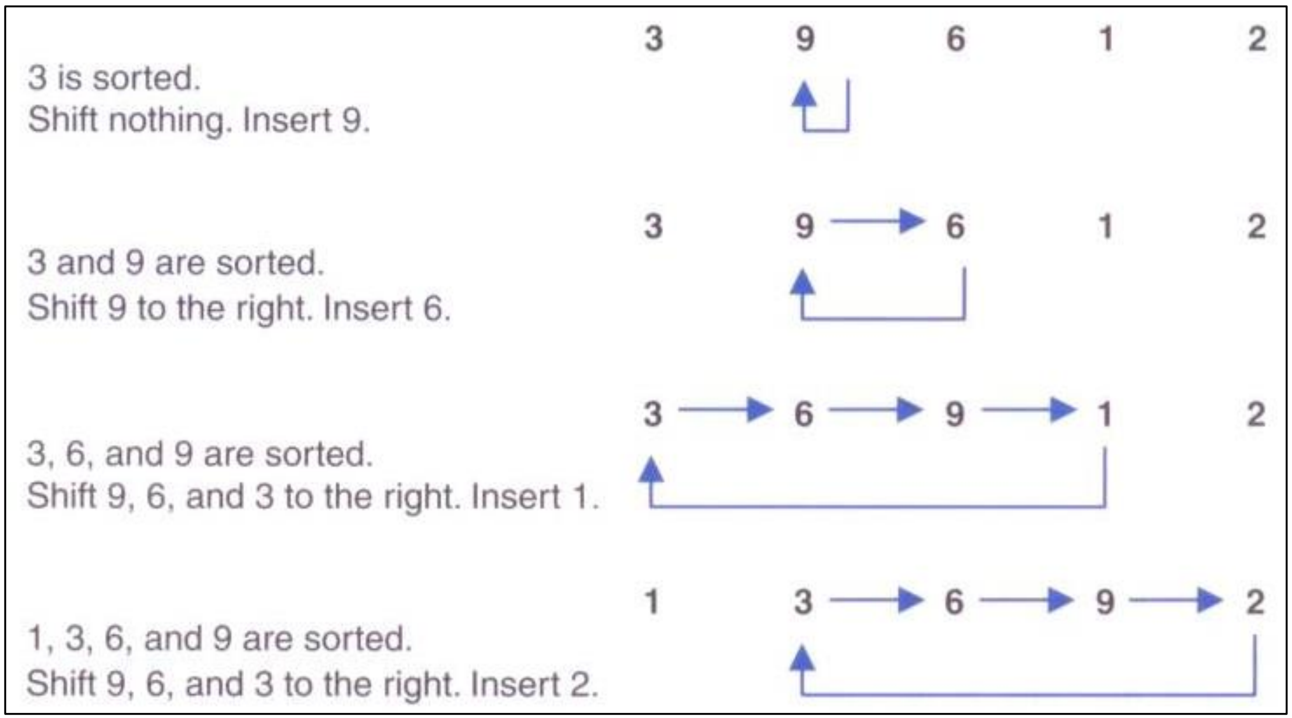
EXPERIENTIA DOCET


Insertion Sort

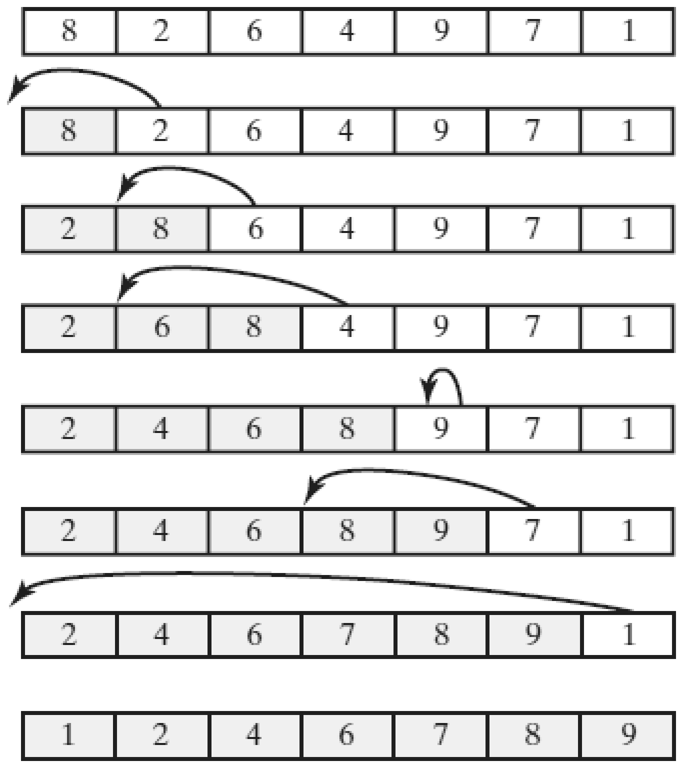
- **Insertion sort** orders values by repetitively **inserting a particular value** into a **sorted subset** of the list
- More specifically:
    1. Consider the **first** item to be a **sorted sublist** of length 1
    2. Insert the **second** item into the **sorted sublist**, shifting the first item if needed
    3. Insert the **third** item into the **sorted sublist**, shifting the other items as needed
    4. Repeat **until all values have been inserted into their proper positions**

**Sorted Sublist**

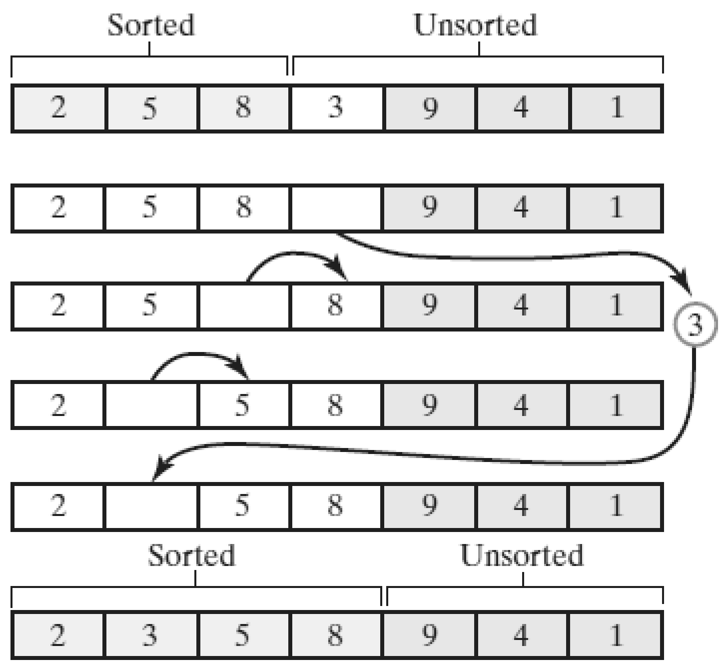
1. Consider the **first** item to be a **sorted sublist** of length 1
2. Insert the **second** item into the **sorted sublist**, shifting the first item if needed
3. Insert the **third** item into the **sorted sublist**, shifting the other items as needed
4. Repeat **until all values have been inserted into their proper positions**



# INSERTION SORT



# INSERTION SORT



```java
    // Insert anEntry, the next unsorted entry, in the sorted sublist
    //
    private static <T extends Comparable<? super T>>
            void insertInOrder(T anEntry, T[] a, int begin, int end) {

        int index = end; // End of the sublist

        // Keep scanning the sublist from end to beginning
        // Compare anEntry vs. last entry (a[index])
        // Shift a[index] towards the end to make room for anEntry
        //
        while ((index >= begin) && (anEntry.compareTo(a[index]) < 0)) {
            a[index + 1] = a[index];                // Make room
            index--;                                // Next left entry
        } // end for

        a[index + 1] = anEntry;  // Insert
    }

```

```java
    public static <T extends Comparable<? super T>>
            void insertionSort(T[] a, int n) {

        insertionSort(a, 0, n - 1);
    }

    public static <T extends Comparable<? super T>>
            void insertionSort(T[] a, int first, int last) {

        // Start at position 1, not 0
        //
        for (int unsorted = first + 1; unsorted <= last; unsorted++) {

            T firstUnsorted = a[unsorted]; // first of Unsorted

            insertInOrder(firstUnsorted, a, first, unsorted - 1);
        }
    }

```

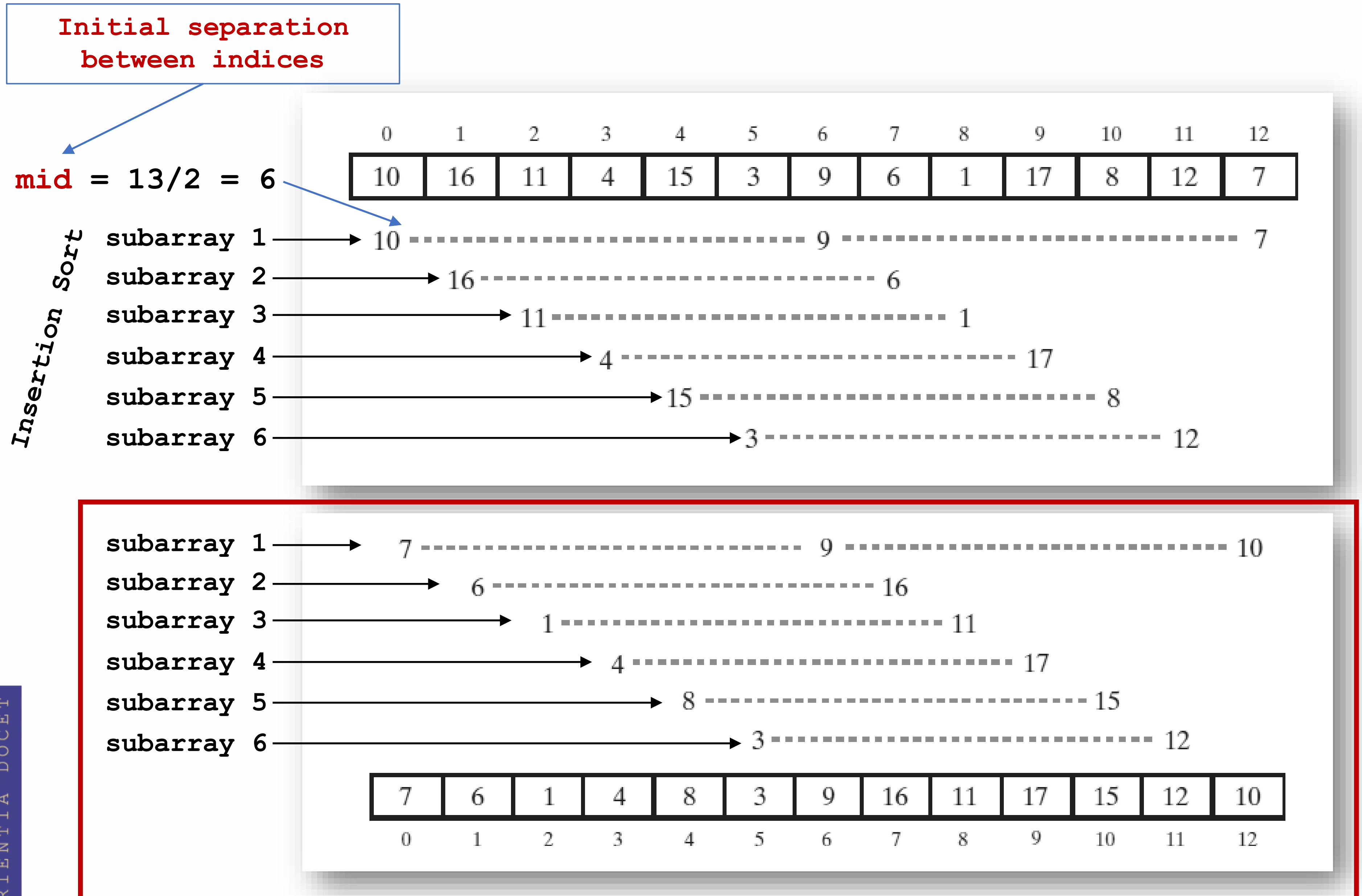
# SHELL SORT


Shell Sort

- **Shell Sort** is a variation of the Insertion Sort which is faster than $O(n^2)$
- Observation during Insertion Sort:
  - When an entry is far from its correct sorted position → **many moves**
  - When an array is completely scrambled → Insertion Sort: **extremely inefficient**
  - When an array is almost sorted → Insertion Sort = Shell Sort: **more efficient**
  - Donald Shell in 1959:
    - **(Insertion) sort subarrays of entries** at **equally spaced indices**.
      - **Initial separation between indices be n/2**
      - **Sort each subarrays separately using insertion sort**
      - **Halve this value each pass**
      - **Sort each subarrays separately using insertion sort**
      - **Repeat** until the **separation is 1**
      - **Final step: Insertion sort the entire array.**

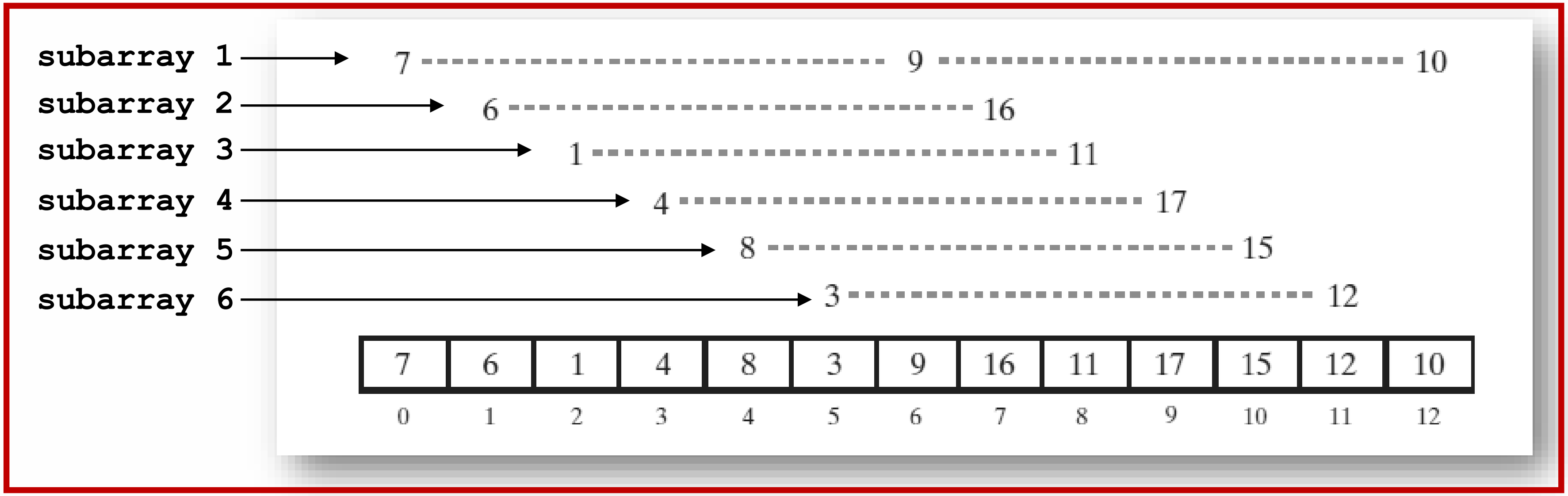
**Almost Sorted n/2**

# Shell Sort



Initial separation between indices
mid = 13/2 = 6
Insertion Sort
0 1 2 3 4 5 6 7 8 9 10 11 12
10 16 11 4 15 3 9 6 1 17 8 12 7
subarray 1 → 10 9 7
subarray 2 → 16 6
subarray 3 → 11 1
subarray 4 → 4 17
subarray 5 → 15 8
subarray 6 → 3 12
subarray 1 → 7 9 10
subarray 2 → 6 16
subarray 3 → 1 11
subarray 4 → 4 17
subarray 5 → 8 15
subarray 6 → 3 12
7 6 1 4 8 3 9 16 11 17 15 12 10
0 1 2 3 4 5 6 7 8 9 10 11 12

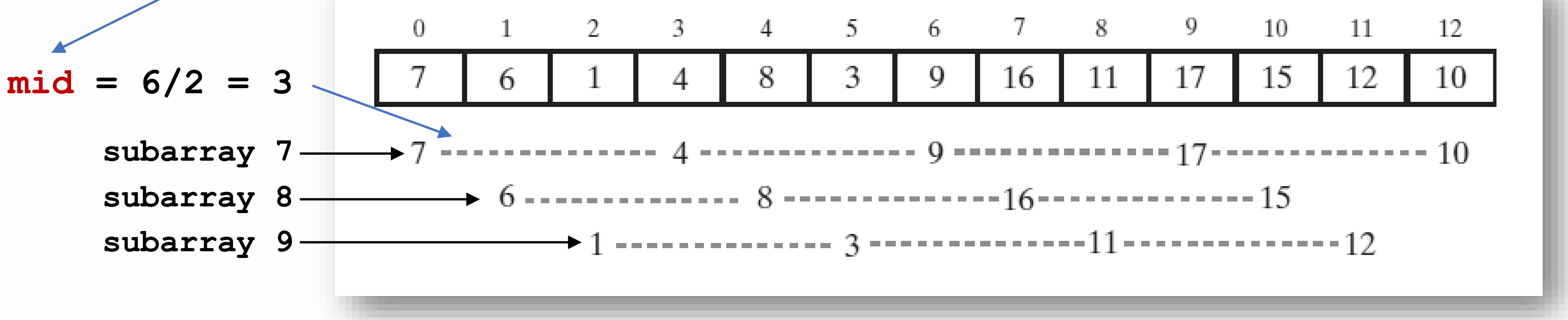
# Shell Sort

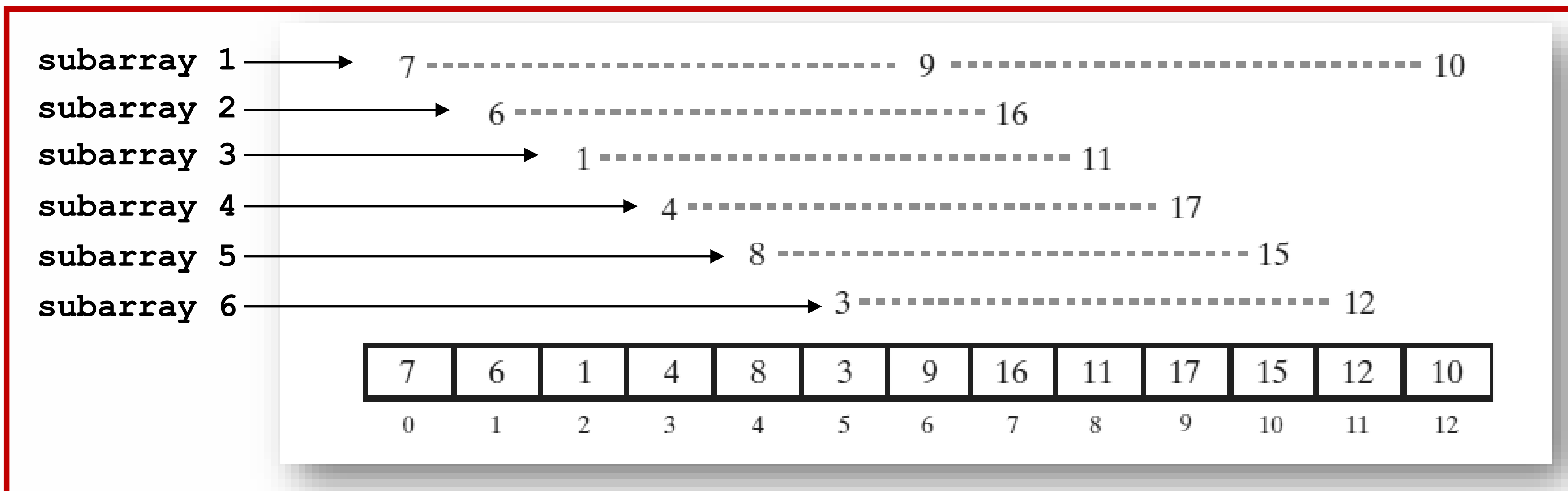


subarray 1
subarray 2
subarray 3
subarray 4
subarray 5
subarray 6
7
9
10
6
16
1
11
4
17
8
15
3
12
7 6 1 4 8 3 9 16 11 17 15 12 10
0 1 2 3 4 5 6 7 8 9 10 11 12

# Shell Sort

2rd round, separation between indices



**2rd round, separation between indices**

**mid = 6/2 = 3**

*Insertion Sort*

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 7 | 6 | 1 | 4 | 8 | 3 | 9 | 16 | 11 | 17 | 15 | 12 | 10 |

subarray 7 → 7 - - - 4 - - - 9 - - - 17 - - - 10

subarray 8 → 6 - - - 8 - - - 16 - - - 15

subarray 9 → 1 - - - 3 - - - 11 - - - 12

subarray 7 → 4 - - - 7 - - - 9 - - - 10 - - - 17

subarray 8 → 6 - - - 8 - - - 15 - - - 16

subarray 9 → 1 - - - 3 - - - 11 - - - 12

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 4 | 6 | 1 | 7 | 8 | 3 | 9 | 15 | 11 | 10 | 16 | 12 | 17 |

3rd round, separation between indices

**mid** = 3/2 = 1

Insertion Sort

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 4 | 6 | 1 | 7 | 8 | 3 | 9 | 15 | 11 | 10 | 16 | 12 | 17 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |

3rd round, separation between indices

**mid = 3/2 = 1**

Insertion Sort

| 4 | 6 | 1 | 7 | 8 | 3 | 9 | 15 | 11 | 10 | 16 | 12 | 17 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |

**Sorted Sublist**

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 3 | 4 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 15 | 16 | 17 |

# Shell Sort

```java
//  Sorts equally spaced elements of an array into ascending order.
//  Parameters:
//     a       An array of Comparable objects.
//     first   The integer index of the first array entry to
//             consider; first >= 0 and < a.length.
//     last    The integer index of the last array entry to
//             consider; last >= first and < a.length.
//     space   The difference between the indices of the
//             entries to sort.
private static <T extends Comparable<? super T>>
        void incrementalInsertionSort(T[] a, int first, int last, int space) {

    int unsorted, index;

    for (unsorted = first + space; unsorted <= last;
            unsorted = unsorted + space) {

        T nextToInsert = a[unsorted];
        index = unsorted - space;

        while ((index >= first) && (nextToInsert.compareTo(a[index]) < 0)) {
            a[index + space] = a[index];
            index = index - space;
        }

        a[index + space] = nextToInsert;
    }
}
```

# SHELL SORT

```java
    public static <T extends Comparable<? super T>>
            void shellSort(T[] a, int n) {

        shellSort(a, 0, n - 1);

    } // end shellSort

    /**
     * Sorts equally spaced elements of an array into ascending order.
     *
     * @param a An array of Comparable objects.
     * @param first An integer >= 0 that is the index of the first array element
     * to consider.
     * @param last An integer >= first and < a.length that is the index of the
     * last array element to consider. @param space The difference between the
     * indices of the eleme nts to sort.
     */
    public static <T extends Comparable<? super T>>
            void shellSort(T[] a, int first, int last) {

        int n = last - first + 1; // Number of array entries
        int space = n / 2;

        while (space > 0) {
            for (int begin = first; begin < first + space; begin++) {
                incrementalInsertionSort(a, begin, last, space);
            }
            space = space / 2;
        }
    }
```

# MORE EXAMPLES

EXPERIENTIA DOCET

**7 5 2 9 1 1 4 3 6**

Show the contents of the array above **each time** an **Selection Sort** changes it while sorting the array into **ascending order**.

Smallest → Final Position

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 7. | 5 | 2 | 9 | 1. | 1 | 4 | 3 | 6 |
| | 1 | 5. | 2 | 9 | 7 | 1. | 4 | 3 | 6 |
| | ✓ | 1 | 2✓ | 9 | 7 | 5 | 4 | 3 | 6 |
| | | ✓ | 2 | 9. | 7 | 5 | 4 | 3. | 6 |
| | | | ✓ | 3 | 7. | 5 | 4. | 9 | 6 |
| | | | | ✓ | 4 | 5✓ | 7 | 9 | 6 |
| | | | | | ✓ | 5 | 7. | 9 | 6. |
| | | | | | | ✓ | 6 | 9. | 7. |
| | | | | | | | ✓ | 7 | 9 |
| | | | | | | | | ✓ | ✓ |
| ✓ | 1 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 9 |
| | | | | | | | | | |

**7 5 2 9 1 1 4 3 6**

Show the contents of the array above **each time** an **Insertion Sort** changes it while sorting the array into **ascending order.**

Sorted [Sublist ]

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| [7] | 5 | 2 | 9 | 1 | 1 | 4 | 3 | 6 |
| [7. | 5.] | | | | | | | |
| [5. | 7 | 2.] | | | | | | |
| [2 | 5 | 7 | 9] | | | | | |
| [2. | 5 | 7 | 9 | 1.] | | | | |
| [1 | 2. | 5 | 7 | 9 | 1.] | | | |
| [1 | 1 | 2 | 5. | 7 | 9 | 4.] | | |
| [1 | 1 | 2 | 4. | 5 | 7 | 9 | 3.] | |
| [1 | 1 | 2 | 3 | 4 | 5 | 7. | 9 | 6.] |
| 1 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 9 |
| | | | | | | | | |

**8 5 3 2 2 9 3 7 4 6** Almost Sorted n/2

Show the contents of the array above **each time a Shell Sort** changes it while sorting the array into **ascending order**.

| | | | | | | | | | | | mid | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 8 | 5 | 3 | 2 | 2 | 9 | 3 | 7 | 4 | 6 | | 10/2=5 | |
| 8. | | | | | 9. | | | | | | | |
| | 3. | | | | | 5. | | | | | | |
| | | 3. | | | | | 7. | | | | | |
| | | | 2. | | | | | 4. | | | | |
| | | | | 2. | | | | | 6. | | | |
| 8 | 3 | 3 | 2 | 2 | 9 | 5 | 7 | 4 | 6 | | 5/2=2 | |
| 2. | | 3. | | 4. | | 5. | | 8. | | | | |
| | 2. | | 3. | | 6. | | 7. | | 9. | | | |
| [2] | 2 | 3 | 3 | 4 | 6 | 5 | 7 | 8 | 9 | | 2/2=1 | * |
| [2 | 2] | | | | | | | | | | | |
| [2 | 2 | 3] | | | | | | | | | | |
| [2 | 2 | 3 | 3] | | | | | | | | | |
| [2 | 2 | 3 | 3 | 4] | | | | | | | | |
| [2 | 2 | 3 | 3 | 4 | 6] | | | | | | | |
| [2 | 2 | 3 | 3 | 4 | 6. | 5.] | | | | | | |
| [2 | 2 | 3 | 3 | 4 | 5 | 6 | 7] | | | | | |
| [2 | 2 | 3 | 3 | 4 | 5 | 6 | 7 | 8] | | | | |
| [2 | 2 | 3 | 3 | 4 | 5 | 6 | 7 | 8 | 9] | | | |
| 2 | 2 | 3 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | | | ✓ |

8 5 3 2 2 9 3 7 4 Almost Sorted n/2

Show the contents of the array above **each time a Shell Sort** changes it while sorting the array into **ascending order**.

need

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | 8 | 5 | 3 | 2 | 2 | 9 | 3 | 7 | 4 | | 9/2 = 4 |
| | 2. | | | | 4. | | | | 8. | | |
| | | 5. | | | | 9. | | | | | |
| | | | 3. | | | | 3. | | | | |
| | | | | 2. | | | | 7. | | | |
| | 2 | 5 | 3 | 2 | 4 | 9 | 3 | 7 | 8 | | 4/2 = 2 |
| | 2. | | 3. | | 3. | | 4. | | 8. | | |
| | | 2. | | 5. | | 7. | | 9. | | | |
| * | [2] | 2 | 3 | 5 | 3 | 7 | 4 | 9 | 8 | | 2/2 = 1 |
| | [2 | 2] | | | | | | | | | |
| | [2 | 2 | 3] | | | | | | | | |
| | [2 | 2 | 3 | 5] | | | | | | | |
| | [2 | 2 | 3 | 5. | 3.] | | | | | | |
| | [2 | 2 | 3 | 3 | 5 | 7] | | | | | |
| | [2 | 2 | 3 | 3 | 5. | 7 | 4.] | | | | |
| | [2 | 2 | 3 | 3 | 4 | 5 | 7 | 9] | | | |
| | [2 | 2 | 3 | 3 | 4 | 5 | 7 | 9. | 8.] | | |
| ✓ | 2 | 2 | 3 | 3 | 4 | 5 | 7 | 8 | 9 | | |
| | | | | | | | | | | | |

9 5 6 4 3 9 3 8 2 Almost Sorted n/2

Show the contents of the array above **each time** a **Shell Sort** changes it while sorting the array into **ascending order**.

| | | | | | | | | | | mid | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 9 | 5 | 6 | 4 | 3 | 9 | 3 | 8 | 2 | | 9/2 = 4 | |
| 2. | | | | 3. | | | | 9. | | | |
| | 5. | | | | 9. | | | | | | |
| | | 3. | | | | 6. | | | | | |
| | | | 4. | | | | 8. | | | | |
| 2 | 5 | 3 | 4 | 3 | 9 | 6 | 8 | 9 | | 4/2 = 2 | |
| 2. | | 3. | | 3. | | 6. | | 9. | | | |
| | 4. | | 5. | | 8. | | 9. | | | | |
| [2] | 4 | 3 | 5 | 3 | 8 | 6 | 9 | 9 | | 2/2 = 1 | * |
| [2 | 4] | | | | | | | | | | |
| [2 | 4. | 3.] | | | | | | | | | |
| [2 | 3 | 4 | 5] | | | | | | | | |
| [2 | 3 | 4. | 5 | 3.] | | | | | | | |
| [2 | 3 | 3 | 4 | 5 | 8] | | | | | | |
| [2 | 3 | 3 | 4 | 5 | 8. | 6.] | | | | | |
| [2 | 3 | 3 | 4 | 5 | 6 | 8 | 9] | | | | |
| [2 | 3 | 3 | 4 | 5 | 6 | 8 | 9 | 9] | | | |
| 2 | 3 | 3 | 4 | 5 | 6 | 8 | 9 | 9 | | | |

# Compare the Sorting Algorithms

```java
public static <T extends Comparable<? super T>> void selectionSort(T[] a, int n) {
        for (int index = 0; index < n - 1; index++) {
            int indexOfNextSmallest = getIndexOfSmallest(a, index, n - 1);
            swap(a, index, indexOfNextSmallest);
        }
}
```

**n – 1, for-loop**

Worst case, **O(n)**

Always 3 ops, **O(1)**

$(n-1) * (O(n) + O(1)) = (n-1) * O(n) = O(n^2)$

**simplify ->** **simplify ->**

swap():

Always 3 steps → Number of Operations is independent of number of entries in an array → **O(1)**

Data movement, **O(n)** swap, little movement.

```java
private static void swap(Object[] a, int i, int j) {
    Object temp = a[i];
    a[i] = a[j];
    a[j] = temp;
}
```

```
private static <T extends Comparable<? super T>>
        int getIndexOfSmallest(T[] a, int first, int last) {

    T min = a[first];
    int indexOfMin = first;

    for (int index = first + 1; index <= last; index++) {

        if (a[index].compareTo(min) < 0) {
            min = a[index];
            indexOfMin = index;
        }
    }

    return indexOfMin;
}
```

getIndexOfSmallest():

last = n – 1

first ranges from 0 to n-2

Each time the 2 loops execute last – first. Worst (n - 1) – 0 = n - 1. Best (n - 1) – (n - 2) = 1.

Then total: (n - 1) + (n - 2) + … + 1 = n (n – 1) / 2 = ½ $\mathbf{n^2}$ – ½ **n**

The selection sort then is $\mathbf{O(n^2)}$

**or**

The worst: **O (n)** for n comparisons, 1 basic operation compareTo() each time.

Outer for-loop runs n -1 so the selection sort then is (n – 1) * $\mathbf{O(n) = O(n^2)}$

EXPERIENTIA DOCET

```
public static <T extends Comparable<? super T>>
        void insertionSort(T[] a, int n) {

    insertionSort(a, 0, n - 1);
}

public static <T extends Comparable<? super T>>
        void insertionSort(T[] a, int first, int last) {

    // Start at position 1, not 0
    //
    for (int unsorted = first + 1; unsorted <= last; unsorted++) {

        T firstUnsorted = a[unsorted]; // first of Unsorted

        insertInOrder(firstUnsorted, a, first, unsorted - 1);
    }
}
```

**n – 1, for loop**

Worst case, **O(n)**

**Worst Case: (n – 1) * O(n) = O($n^2$)**
**Best Case: (n – 1) * O(1) = O(n)**

EXPERIENTIA DOCET

```java
private static <T extends Comparable<? super T>>
        void insertInOrder(T anEntry, T[] a, int begin, int end) {

    int index = end; // End of the sublist

    // Keep scanning the sublist from end to beginning
    // Compare anEntry vs. last entry (a[index])
    // Shift a[index] towards the end to make room for anEntry
    //
    while ((index >= begin) && (anEntry.compareTo(a[index]) < 0)) {
        a[index + 1] = a[index];               // Make room
        index--;                               // Next left entry
    } // end for

    a[index + 1] = anEntry;  // Insert
}
```

insertInOrder():

begin is 0

end ranges from 0 to n – 2

1 basic operation compareTo() each time.

Worst case O(n). Best case O(1).

Total: 1 + 2 + … + (n-1) = n(n-1)/2

Thus, insertion sort is **O(n²)** ← worst case

When array already sorted, **O(n)** ← best case

```
public static <T extends Comparable<? super T>>
        void shellSort(T[] a, int first, int last) {

    int n = last - first + 1; // Number of array entries
    int space = n / 2;

    while (space > 0) {
        for (int begin = first; begin < first + space; begin++) {
  1         incrementalInsertionSort(a, begin, last, space);
        }
        space = space / 2;
    }
}
```

```
private static <T extends Comparable<? super T>>
        void incrementalInsertionSort(T[] a, int first, int last, int space) {

    int unsorted, index;

    for (unsorted = first + space; unsorted <= last;
 2          unsorted = unsorted + space) {

        T nextToInsert = a[unsorted];
        index = unsorted - space;

        while ((index >= first) && (nextToInsert.compareTo(a[index]) < 0)) {
  3         a[index + space] = a[index];
            index = index - space;
        }

        a[index + space] = nextToInsert;
    }
}
```

**O(n³),** 3 nested loops
but space n/2 each time
**O(n^1.5)** is the worst-case of Shell
**O(n)** is the best, an ordinary insertion sort

| Time efficiencies in Big Oh notation | | | |
|---|---|---|---|
| | Best | Average | Worst |
| Selection Sort | **$O(n^2)$** | **$O(n^2)$** | **$O(n^2)$** |
| Insertion Sort | **$O(n)$** | **$O(n^2)$** | **$O(n^2)$** |
| Shell Sort | **$O(n)$** | **$O(n^{1.5})$** | **$O(n^{1.5})$** |

## Array Sorting Algorithms

| Algorithm | Time Complexity | | | Space Complexity |
|---|---|---|---|---|
| | Best | Average | Worst | Worst |
| Quicksort | Ω(n log(n)) | Θ(n log(n)) | O(n^2) | O(log(n)) |
| Mergesort | Ω(n log(n)) | Θ(n log(n)) | O(n log(n)) | O(n) |
| Timsort | Ω(n) | Θ(n log(n)) | O(n log(n)) | O(n) |
| Heapsort | Ω(n log(n)) | Θ(n log(n)) | O(n log(n)) | O(1) |
| Bubble Sort | Ω(n) | Θ(n^2) | O(n^2) | O(1) |
| Insertion Sort | Ω(n) | Θ(n^2) | O(n^2) | O(1) |
| Selection Sort | Ω(n^2) | Θ(n^2) | O(n^2) | O(1) |
| Tree Sort | Ω(n log(n)) | Θ(n log(n)) | O(n^2) | O(n) |
| Shell Sort | Ω(n log(n)) | Θ(n(log(n))^2) | O(n(log(n))^2) | O(1) |
| Bucket Sort | Ω(n+k) | Θ(n+k) | O(n^2) | O(n) |

| | Best Case | Average Case | Worst Case |
|---|---|---|---|
| Selection sort | $O(n^2)$ | $O(n^2)$ | $O(n^2)$ |
| Insertion sort | $O(n)$ | $O(n^2)$ | $O(n^2)$ |
| Shell sort | $O(n)$ | $O(n^{1.5})$ | $O(n^2)$ or $O(n^{1.5})$ |

Horrible | Bad | Fair | Good | Excellent

http://bigocheatsheet.com/

# Array Sorting Algorithms

| Algorithm | Time Complexity | | | Space Complexity |
|---|---|---|---|---|
| | Best | Average | Worst | Worst |
| Quicksort | Ω(n log(n)) | Θ(n log(n)) | O(n^2) | O(log(n)) |
| Mergesort | Ω(n log(n)) | Θ(n log(n)) | O(n log(n)) | O(n) |
| Timsort | Ω(n) | Θ(n log(n)) | O(n log(n)) | O(n) |
| Heapsort | Ω(n log(n)) | Θ(n log(n)) | O(n log(n)) | O(1) |
| Bubble Sort | Ω(n) | Θ(n^2) | O(n^2) | O(1) |
| Insertion Sort | Ω(n) | Θ(n^2) | O(n^2) | O(1) |
| Selection Sort | Ω(n^2) | Θ(n^2) | O(n^2) | O(1) |
| Tree Sort | Ω(n log(n)) | Θ(n log(n)) | O(n^2) | O(n) |
| Shell Sort | Ω(n log(n)) | Θ(n(log(n))^2) | O(n(log(n))^2) | O(1) |
| Bucket Sort | Ω(n+k) | Θ(n+k) | O(n^2) | O(n) |
| Radix Sort | Ω(nk) | Θ(nk) | O(nk) | O(n+k) |
| Counting Sort | Ω(n+k) | Θ(n+k) | O(n+k) | O(k) |
| Cubesort | Ω(n) | Θ(n log(n)) | O(n log(n)) | O(n) |

http://bigocheatsheet.com/

Horrible Bad Fair Good Excellent

*See you next class!*