SORTING TECHNIQUES

By:
Dr. Sushil Kumar
Assistant Professor
Department of Computer Sc. & Engg.
National Institute of Technology Warangal

Sorting

- Bubble Sort
- Selection Sort
- Insertion Sort
- Merge Sort
- Quick Sort
- Bucket Sort

Sorting

- Bubble Sort
- Selection Sort
- Insertion Sort
- Merge Sort
- Quick Sort
- Bucket Sort

Bubble Sort

- **Bubble sort**: orders a list of values by repetitively comparing neighboring elements and swapping their positions if necessary

Bubble Sort

- **Bubble sort**: orders a list of values by repetitively comparing neighboring elements and swapping their positions if necessary.

more specifically:

- scan the list, exchanging adjacent elements if they are not in relative order; this bubbles the highest value to the top
- scan the list again, bubbling up the second highest value
- repeat until all elements have been placed in their proper order

Bubble Sort

- **Bubble sort**: orders a list of values by repetitively comparing neighboring elements and swapping their positions if necessary

0	1	2	3	4	5
77	42	35	12	101	5

Bubble Sort

- **Bubble sort**: orders a list of values by repetitively comparing neighboring elements and swapping their positions if necessary
- Iteration- 1

0	1	2	3	4	5
77	42	35	12	101	5

Bubble Sort

- **Bubble sort**: orders a list of values by repetitively comparing neighboring elements and swapping their positions if necessary
- Iteration- 1

0	1	2	3	4	5
77	42	35	12	101	5

>

Bubble Sort

- **Bubble sort**: orders a list of values by repetitively comparing neighboring elements and swapping their positions if necessary
- Iteration- 1

0	1	2	3	4	5
42	77	35	12	101	5

Bubble Sort

- **Bubble sort**: orders a list of values by repetitively comparing neighboring elements and swapping their positions if necessary
- Iteration- 1

0	1	2	3	4	5
42	77	35	12	101	5

Bubble Sort

- **Bubble sort**: orders a list of values by repetitively comparing neighboring elements and swapping their positions if necessary
- Iteration- 1

0	1	2	3	4	5
42	77	35	12	101	5

>

Bubble Sort

- **Bubble sort**: orders a list of values by repetitively comparing neighboring elements and swapping their positions if necessary
- Iteration- 1

0	1	2	3	4	5
42	35	77	12	101	5

Bubble Sort

- **Bubble sort**: orders a list of values by repetitively comparing neighboring elements and swapping their positions if necessary
- Iteration- 1

0	1	2	3	4	5
42	35	77	12	101	5

Bubble Sort

- **Bubble sort**: orders a list of values by repetitively comparing neighboring elements and swapping their positions if necessary
- Iteration- 1

0	1	2	3	4	5
42	35	12	77	101	5

Bubble Sort

- **Bubble sort**: orders a list of values by repetitively comparing neighboring elements and swapping their positions if necessary
- Iteration- 1

0	1	2	3	4	5
42	35	12	77	101	5

Bubble Sort

- **Bubble sort**: orders a list of values by repetitively comparing neighboring elements and swapping their positions if necessary
- Iteration- 1

0	1	2	3	4	5
42	35	12	77	101	5

Bubble Sort

- **Bubble sort**: orders a list of values by repetitively comparing neighboring elements and swapping their positions if necessary
- Iteration- 1

0	1	2	3	4	5
42	35	12	77	5	101

Bubble Sort

- **Bubble sort**: orders a list of values by repetitively comparing neighboring elements and swapping their positions if necessary
- Iteration- 1

0	1	2	3	4	5
42	35	12	77	5	101

Bubble Sort

- **Bubble sort**: orders a list of values by repetitively comparing neighboring elements and swapping their positions if necessary
- Iteration- 1

0	1	2	3	4	5
42	35	12	77	5	101

Bubble Sort

- **Bubble sort**: orders a list of values by repetitively comparing neighboring elements and swapping their positions if necessary
- Iteration- 2

0	1	2	3	4	5
42	35	12	77	5	101

Bubble Sort

- **Bubble sort**: orders a list of values by repetitively comparing neighboring elements and swapping their positions if necessary
- Iteration- 2

0	1	2	3	4	5
35	42	12	77	5	101

Bubble Sort

- **Bubble sort**: orders a list of values by repetitively comparing neighboring elements and swapping their positions if necessary
- Iteration- 2

0	1	2	3	4	5
35	42	12	77	5	101

Bubble Sort

- **Bubble sort**: orders a list of values by repetitively comparing neighboring elements and swapping their positions if necessary
- Iteration- 2

0	1	2	3	4	5
35	12	42	77	5	101

Bubble Sort

- **Bubble sort**: orders a list of values by repetitively comparing neighboring elements and swapping their positions if necessary
- Iteration- 2

0	1	2	3	4	5
35	12	42	77	5	101

Bubble Sort

- **Bubble sort**: orders a list of values by repetitively comparing neighboring elements and swapping their positions if necessary
- Iteration- 2

0	1	2	3	4	5
35	12	42	77	5	101

Bubble Sort

- **Bubble sort**: orders a list of values by repetitively comparing neighboring elements and swapping their positions if necessary
- Iteration- 2

0	1	2	3	4	5
35	12	42	5	77	101

Bubble Sort

- **Bubble sort**: orders a list of values by repetitively comparing neighboring elements and swapping their positions if necessary
- Iteration- 2

0	1	2	3	4	5
35	12	42	5	77	101

Bubble Sort

- **Bubble sort**: orders a list of values by repetitively comparing neighboring elements and swapping their positions if necessary
- Iteration- 3

0	1	2	3	4	5
35	12	42	5	77	101

Bubble Sort

- **Bubble sort**: orders a list of values by repetitively comparing neighboring elements and swapping their positions if necessary
- Iteration- 3

0	1	2	3	4	5
12	35	42	5	77	101

Bubble Sort

- **Bubble sort**: orders a list of values by repetitively comparing neighboring elements and swapping their positions if necessary
- Iteration- 3

0	1	2	3	4	5
12	35	42	5	77	101

Bubble Sort

- **Bubble sort**: orders a list of values by repetitively comparing neighboring elements and swapping their positions if necessary
- Iteration- 3

0	1	2	3	4	5
12	35	42	5	77	101

Bubble Sort

- **Bubble sort**: orders a list of values by repetitively comparing neighboring elements and swapping their positions if necessary
- Iteration- 3

0	1	2	3	4	5
12	35	42	5	77	101

Bubble Sort

- **Bubble sort**: orders a list of values by repetitively comparing neighboring elements and swapping their positions if necessary
- Iteration- 3

0	1	2	3	4	5
12	35	5	42	77	101

Bubble Sort

- **Bubble sort**: orders a list of values by repetitively comparing neighboring elements and swapping their positions if necessary
- Iteration- 3

0	1	2	3	4	5
12	35	5	42	77	101

Bubble Sort

- **Bubble sort**: orders a list of values by repetitively comparing neighboring elements and swapping their positions if necessary
- Iteration- 4

0	1	2	3	4	5
12	35	5	42	77	101

Bubble Sort

- **Bubble sort**: orders a list of values by repetitively comparing neighboring elements and swapping their positions if necessary
- Iteration- 4

0	1	2	3	4	5
12	35	5	42	77	101

Bubble Sort

- **Bubble sort**: orders a list of values by repetitively comparing neighboring elements and swapping their positions if necessary
- Iteration- 4

0	1	2	3	4	5
12	5	35	42	77	101

Bubble Sort

- **Bubble sort**: orders a list of values by repetitively comparing neighboring elements and swapping their positions if necessary
- Iteration- 4

0	1	2	3	4	5
12	5	35	42	77	101

Bubble Sort

- **Bubble sort**: orders a list of values by repetitively comparing neighboring elements and swapping their positions if necessary
- Iteration- 5

0	1	2	3	4	5
12	5	35	42	77	101

Bubble Sort

- **Bubble sort**: orders a list of values by repetitively comparing neighboring elements and swapping their positions if necessary
- Iteration- 5

0	1	2	3	4	5
5	12	35	42	77	101

Bubble Sort

- **Bubble sort**: orders a list of values by repetitively comparing neighboring elements and swapping their positions if necessary
- Iteration- 5

0	1	2	3	4	5
5	12	35	42	77	101

Bubble Sort Code

```
public static void bubbleSort(int[] a) {
    for (int i = 0; i < a.length; i++) {
        for (int j = 1; j < a.length - i; j++) {
            // swap adjacent out-of-order elements
            if (a[j-1] > a[j]) {
                swap(a, j-1, j);
            }
        }
    }
}
```

Bubble Sort

Running time (# comparisons) for input size *n*:

$$\sum_{i=0}^{n-1}\sum_{j=1}^{n-1-i}1 = \sum_{i=0}^{n-1}(n-1-i)$$

$$= n\sum_{i=0}^{n-1}1 - \sum_{i=0}^{n-1}1 - \sum_{i=0}^{n-1}i$$

$$= n^2 - n - \frac{(n-1)n}{2}$$

$$= \Theta(n^2)$$

number of actual swaps performed depends on the data; out-of-order data performs many swaps

Sorting

- Bubble Sort
- Selection Sort
- Insertion Sort
- Merge Sort
- Quick Sort
- Bucket Sort

Sorting

- Bubble Sort
- Selection Sort
- Insertion Sort
- Merge Sort
- Quick Sort
- Bucket Sort

Selection sort

- **Selection sort**: orders a list of values by repetitively putting a particular value into its final position.

Selection sort

- **Selection sort**: orders a list of values by repetitively putting a particular value into its final position.

more specifically:

- ✓ find the smallest value in the list
- ✓ switch it with the value in the first position
- ✓ find the next smallest value in the list
- ✓ switch it with the value in the second position
- ✓ repeat until all values are in their proper places

Selection sort

- **Selection sort**: orders a list of values by repetitively putting a particular value into its final position.
- Take the 1st Element

0	1	2	3	4	5
77	42	35	12	101	5

Selection sort

- **Selection sort**: orders a list of values by repetitively putting a particular value into its final position.
- Take the 1st Element

0	1	2	3	4	5
77	42	35	12	101	5

Selection sort

- **Selection sort**: orders a list of values by repetitively putting a particular value into its final position.
- Take the 1st Element
- Now search for minimum element, if exist exchange it.

0	1	2	3	4	5
77	42	35	12	101	5

Selection sort

- **Selection sort**: orders a list of values by repetitively putting a particular value into its final position.
- Take the 1st Element
- Now search for minimum element, if exist exchange it.

0	1	2	3	4	5
77	42	35	12	101	5

Selection sort

- **Selection sort**: orders a list of values by repetitively putting a particular value into its final position.
- Take the 1st Element
- Now search for minimum element, if exist exchange it.

0	1	2	3	4	5
77	42	35	12	101	5

Selection sort

- **Selection sort**: orders a list of values by repetitively putting a particular value into its final position.
- Take the 1st Element
- Now search for minimum element, if exist exchange it.

0	1	2	3	4	5
77	42	35	12	101	5

Selection sort

- **Selection sort**: orders a list of values by repetitively putting a particular value into its final position.
- Take the 1st Element
- Now search for minimum element, if exist exchange it.

0	1	2	3	4	5
77	42	35	12	101	5

Selection sort

- **Selection sort**: orders a list of values by repetitively putting a particular value into its final position.
- Take the 1st Element
- Now search for minimum element, if exist exchange it.

0	1	2	3	4	5
77	42	35	12	101	5

Selection sort

- **Selection sort**: orders a list of values by repetitively putting a particular value into its final position.
- Take the 1st Element
- Now search for minimum element, if exist exchange it.

0	1	2	3	4	5
77	42	35	12	101	5

Selection sort

- **Selection sort**: orders a list of values by repetitively putting a particular value into its final position.
- Take the 1st Element
- Now search for minimum element, if exist exchange it.

0	1	2	3	4	5
77	42	35	12	101	5

- EXCHANGE

Selection sort

- **Selection sort**: orders a list of values by repetitively putting a particular value into its final position.
- Take the 1st Element
- Now search for minimum element, if exist exchange it.

0	1	2	3	4	5
5	42	35	12	101	77

- EXCHANGE

Selection sort

- **Selection sort**: orders a list of values by repetitively putting a particular value into its final position.
- Take the 2nd Element. First is sorted now.
- Now search for minimum element, if exist exchange it.

0	1	2	3	4	5
5	42	35	12	101	77

Selection sort

- **Selection sort**: orders a list of values by repetitively putting a particular value into its final position.
- Take the 2nd Element. First is sorted now.
- Now search for minimum element, if exist exchange it.

0	1	2	3	4	5
5	42	35	12	101	77

Selection sort

- **Selection sort**: orders a list of values by repetitively putting a particular value into its final position.
- Take the 2nd Element. First is sorted now.
- Now search for minimum element, if exist exchange it.

0	1	2	3	4	5
5	42	35	12	101	77

Selection sort

- **Selection sort**: orders a list of values by repetitively putting a particular value into its final position.
- Take the 2nd Element. First is sorted now.
- Now search for minimum element, if exist exchange it.

0	1	2	3	4	5
5	42	35	12	101	77

Selection sort

- **Selection sort**: orders a list of values by repetitively putting a particular value into its final position.
- Take the 2nd Element. First is sorted now.
- Now search for minimum element, if exist exchange it.

0	1	2	3	4	5
5	42	35	12	101	77

Selection sort

- **Selection sort**: orders a list of values by repetitively putting a particular value into its final position.
- Take the 2nd Element. First is sorted now.
- Now search for minimum element, if exist exchange it.

0	1	2	3	4	5
5	42	35	12	101	77

Selection sort

- **Selection sort**: orders a list of values by repetitively putting a particular value into its final position.
- Take the 2nd Element. First is sorted now.
- Now search for minimum element, if exist exchange it.

0	1	2	3	4	5
5	42	35	12	101	77

- EXCHANGE

Selection sort

- **Selection sort**: orders a list of values by repetitively putting a particular value into its final position.
- Take the 2nd Element. First is sorted now.
- Now search for minimum element, if exist exchange it.

0	1	2	3	4	5
5	12	35	42	101	77

- EXCHANGE

Selection sort

- **Selection sort**: orders a list of values by repetitively putting a particular value into its final position.
- Take the 2nd Element. First is sorted now.
- Now search for minimum element, if exist exchange it.

0	1	2	3	4	5
5	12	35	42	101	77

Selection sort

- **Selection sort**: orders a list of values by repetitively putting a particular value into its final position.
- Take the 3rd Element. Till Second is sorted now.
- Now search for minimum element, if exist exchange it.

0	1	2	3	4	5
5	12	35	42	101	77

Selection sort

- **Selection sort**: orders a list of values by repetitively putting a particular value into its final position.
- Take the 3rd Element. Till Second is sorted now.
- Now search for minimum element, if exist exchange it.

0	1	2	3	4	5
5	12	35	42	101	77

Selection sort

- **Selection sort**: orders a list of values by repetitively putting a particular value into its final position.
- Take the 3rd Element. Till Second is sorted now.
- Now search for minimum element, if exist exchange it.

0	1	2	3	4	5
5	12	35	42	101	77

Selection sort

- **Selection sort**: orders a list of values by repetitively putting a particular value into its final position.
- Take the 3rd Element. Till Second is sorted now.
- Now search for minimum element, if exist exchange it.

0	1	2	3	4	5
5	12	35	42	101	77

- EXCHANGE NOT REQUIRED

Selection sort

- **Selection sort**: orders a list of values by repetitively putting a particular value into its final position.
- Take the 3rd Element. Till Second is sorted now.
- Now search for minimum element, if exist exchange it.

0	1	2	3	4	5
5	12	35	42	101	77

- EXCHANGE NOT REQUIRED

Selection sort

- **Selection sort**: orders a list of values by repetitively putting a particular value into its final position.
- Take the 4th Element.
- Now search for minimum element, if exist exchange it.

0	1	2	3	4	5
5	12	35	42	101	77

Selection sort

- **Selection sort**: orders a list of values by repetitively putting a particular value into its final position.
- Take the 4th Element.
- Now search for minimum element, if exist exchange it.

0	1	2	3	4	5
5	12	35	42	101	77

Selection sort

- **Selection sort**: orders a list of values by repetitively putting a particular value into its final position.
- Take the 4th Element.
- Now search for minimum element, if exist exchange it.

0	1	2	3	4	5
5	12	35	42	101	77

Selection sort

- **Selection sort**: orders a list of values by repetitively putting a particular value into its final position.
- Take the 4th Element.
- Now search for minimum element, if exist exchange it.

0	1	2	3	4	5
5	12	35	42	101	77

Selection sort

- **Selection sort**: orders a list of values by repetitively putting a particular value into its final position.
- Take the 4th Element.
- Now search for minimum element, if exist exchange it.

0	1	2	3	4	5
5	12	35	42	101	77

- EXCHANGE NOT REQUIRED

Selection sort

- **Selection sort**: orders a list of values by repetitively putting a particular value into its final position.
- Take the 5th Element.
- Now search for minimum element, if exist exchange it.

0	1	2	3	4	5
5	12	35	42	101	77

Selection sort

- **Selection sort**: orders a list of values by repetitively putting a particular value into its final position.
- Take the 5th Element.
- Now search for minimum element, if exist exchange it.

0	1	2	3	4	5
5	12	35	42	101	77

Selection sort

- **Selection sort**: orders a list of values by repetitively putting a particular value into its final position.
- Take the 5th Element.
- Now search for minimum element, if exist exchange it.

0	1	2	3	4	5
5	12	35	42	101	77

- EXCHANGE

Selection sort

- **Selection sort**: orders a list of values by repetitively putting a particular value into its final position.
- Take the 5th Element.
- Now search for minimum element, if exist exchange it.

0	1	2	3	4	5
5	12	35	42	77	101

- EXCHANGE

Selection sort

- **Selection sort**: orders a list of values by repetitively putting a particular value into its final position.
- Take the 5th Element.
- Now search for minimum element, if exist exchange it.

0	1	2	3	4	5
5	12	35	42	77	101

- EXCHANGE

Selection Sort CODE

```
public static void selectionSort(int[] a) {
    for (int i = 0; i < a.length; i++) {
        // find index of smallest element
        int minIndex = i;
        for (int j = i + 1; j < a.length; j++) {
            if (a[j] < a[minIndex]) {
                minIndex = j;
            }
        }

        // swap smallest element with a[i]
        swap(a, i, minIndex);
    }
}
```

Selection Sort Time

Running time for input size *n*:

In practice, a bit faster than bubble sort. Why?

$$\sum_{i=0}^{n-1}\sum_{j=i+1}^{n-1}1 = \sum_{i=0}^{n-1}(n-1-(i+1)+1)$$

$$= \sum_{i=0}^{n-1}(n-i-1)$$

$$= n\sum_{i=0}^{n-1}1 - \sum_{i=0}^{n-1}i - \sum_{i=0}^{n-1}1$$

$$= n^2 - \frac{(n-1)n}{2} - n$$

$$= \Theta(n^2)$$

Sorting

- Bubble Sort
- Selection Sort
- Insertion Sort
- Merge Sort
- Quick Sort
- Bucket Sort

Sorting

- Bubble Sort
- Selection Sort
- Insertion Sort
- Merge Sort
- Quick Sort
- Bucket Sort

Insertion Sort

insertion sort: orders a list of values by repetitively inserting a particular value into a sorted subset of the list

Insertion Sort

insertion sort: orders a list of values by repetitively inserting a particular value into a sorted subset of the list

more specifically:

- ✓ consider the first item to be a sorted sublist of length 1
- ✓ insert the second item into the sorted sublist, shifting the first item if needed
- ✓ insert the third item into the sorted sublist, shifting the other items as needed
- ✓ repeat until all values have been inserted into their proper positions

Insertion sort

Simple sorting algorithm.

- ✓ n-1 passes over the array
- ✓ At the end of pass *i*, the elements that occupied A[0]…A[*i*] originally are still in those spots and in sorted order.

0	1	2	3	4	5
77	42	35	12	101	5

Insertion sort

Simple sorting algorithm.

- ✓ n-1 passes over the array
- ✓ At the end of pass *i*, the elements that occupied A[0]…A[*i*] originally are still in those spots and in sorted order.

0	1	2	3	4	5
77	42	35	12	101	5

Insertion sort

Simple sorting algorithm.

- ✓ n-1 passes over the array
- ✓ At the end of pass *i*, the elements that occupied A[0]…A[*i*] originally are still in those spots and in sorted order.

0	1	2	3	4	5
77	42	35	12	101	5

Insertion sort

Simple sorting algorithm.

- ✓ n-1 passes over the array
- ✓ At the end of pass *i*, the elements that occupied A[0]…A[*i*] originally are still in those spots and in sorted order.

0	1	2	3	4	5
77	42	35	12	101	5

Insertion sort

Simple sorting algorithm.

- ✓ n-1 passes over the array
- ✓ At the end of pass *i*, the elements that occupied A[0]…A[*i*] originally are still in those spots and in sorted order.

0	1	2	3	4	5
77	42	35	12	101	5

Insertion sort

Simple sorting algorithm.

- ✓ n-1 passes over the array
- ✓ At the end of pass *i*, the elements that occupied A[0]…A[*i*] originally are still in those spots and in sorted order.

0	1	2	3	4	5
77	42	35	12	101	5

Insertion sort

Simple sorting algorithm.

- ✓ n-1 passes over the array
- ✓ At the end of pass *i*, the elements that occupied A[0]…A[*i*] originally are still in those spots and in sorted order.

0	1	2	3	4	5
77	42	35	12	101	

5

Insertion sort

Simple sorting algorithm.

- ✓ n-1 passes over the array
- ✓ At the end of pass *i*, the elements that occupied A[0]…A[*i*] originally are still in those spots and in sorted order.

0	1	2	3	4	5
77	42	35	12		101

5

Insertion sort

Simple sorting algorithm.

- ✓ n-1 passes over the array
- ✓ At the end of pass *i*, the elements that occupied A[0]…A[*i*] originally are still in those spots and in sorted order.

0	1	2	3	4	5
77	42	35		12	101

5

Insertion sort

Simple sorting algorithm.

- ✓ n-1 passes over the array
- ✓ At the end of pass *i*, the elements that occupied A[0]…A[*i*] originally are still in those spots and in sorted order.

0	1	2	3	4	5
77	42		35	12	101

5

Insertion sort

Simple sorting algorithm.

- ✓ n-1 passes over the array
- ✓ At the end of pass *i*, the elements that occupied A[0]…A[*i*] originally are still in those spots and in sorted order.

0	1	2	3	4	5
77		42	35	12	101

5

Insertion sort

Simple sorting algorithm.

- ✓ n-1 passes over the array
- ✓ At the end of pass *i*, the elements that occupied A[0]…A[*i*] originally are still in those spots and in sorted order.

0	1	2	3	4	5
	77	42	35	12	101

5

Insertion sort

Simple sorting algorithm.

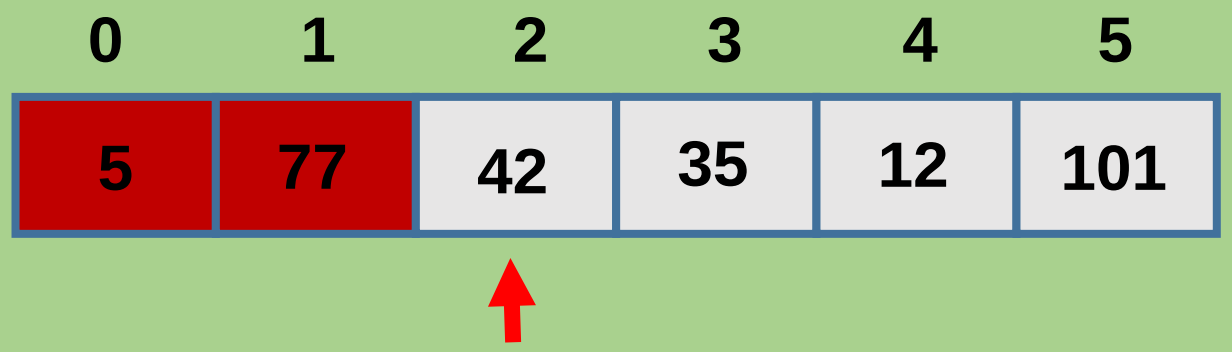
- ✓ n-1 passes over the array
- ✓ At the end of pass *i*, the elements that occupied A[0]…A[*i*] originally are still in those spots and in sorted order.

0	1	2	3	4	5
5	77	42	35	12	101

Insertion sort

Simple sorting algorithm.

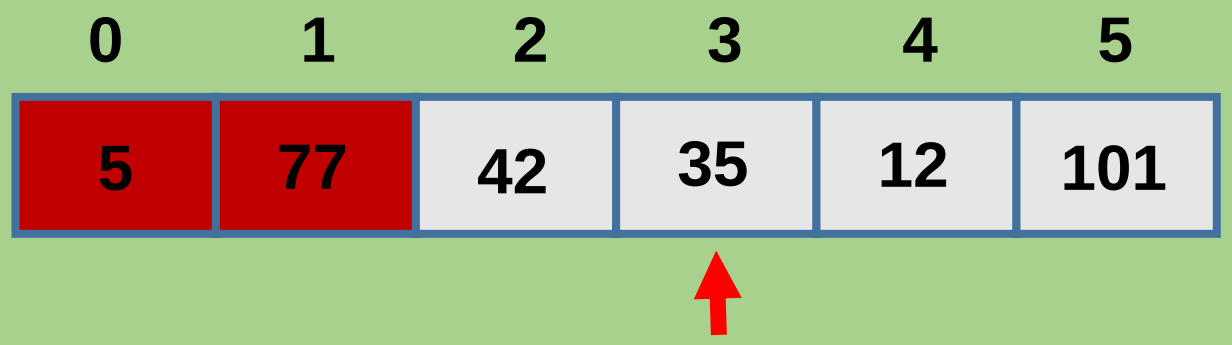
- ✓ n-1 passes over the array
- ✓ At the end of pass *i*, the elements that occupied A[0]…A[*i*] originally are still in those spots and in sorted order.



Insertion sort

Simple sorting algorithm.

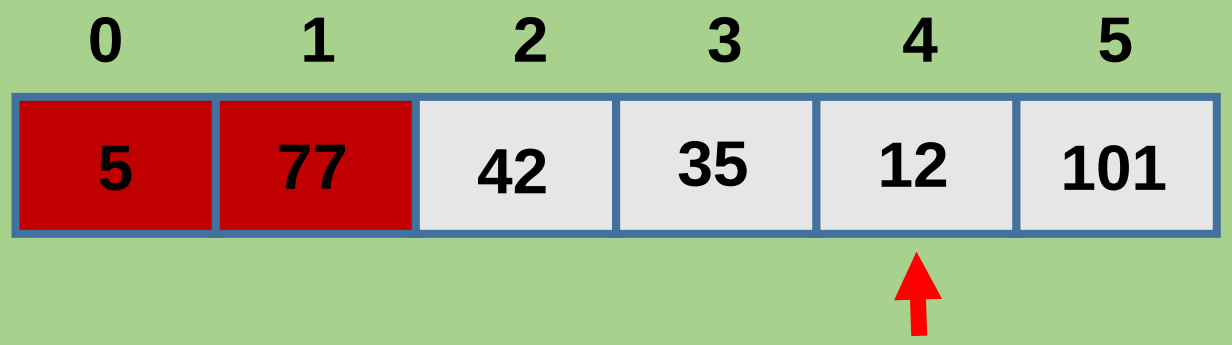
- ✓ n-1 passes over the array
- ✓ At the end of pass *i*, the elements that occupied A[0]…A[*i*] originally are still in those spots and in sorted order.



Insertion sort

Simple sorting algorithm.

- ✓ n-1 passes over the array
- ✓ At the end of pass *i*, the elements that occupied A[0]…A[*i*] originally are still in those spots and in sorted order.



Insertion sort

Simple sorting algorithm.

- ✓ n-1 passes over the array
- ✓ At the end of pass *i*, the elements that occupied A[0]…A[*i*] originally are still in those spots and in sorted order.

0	1	2	3	4	5
5	77	42	35	12	101

Insertion sort

Simple sorting algorithm.

- ✓ n-1 passes over the array
- ✓ At the end of pass *i*, the elements that occupied A[0]…A[*i*] originally are still in those spots and in sorted order.

0	1	2	3	4	5
5	77	42	35	12	101

Insertion sort

Simple sorting algorithm.

- ✓ n-1 passes over the array
- ✓ At the end of pass *i*, the elements that occupied A[0]…A[*i*] originally are still in those spots and in sorted order.

0	1	2	3	4	5
5	77	42	35		101

12

Insertion sort

Simple sorting algorithm.

- ✓ n-1 passes over the array
- ✓ At the end of pass *i*, the elements that occupied A[0]…A[*i*] originally are still in those spots and in sorted order.

0	1	2	3	4	5
5	77	42		35	101

12

Insertion sort

Simple sorting algorithm.

- ✓ n-1 passes over the array
- ✓ At the end of pass *i*, the elements that occupied A[0]…A[*i*] originally are still in those spots and in sorted order.

0	1	2	3	4	5
5	77		42	35	101

12

Insertion sort

Simple sorting algorithm.

- ✓ n-1 passes over the array
- ✓ At the end of pass *i*, the elements that occupied A[0]…A[*i*] originally are still in those spots and in sorted order.

0	1	2	3	4	5
5		77	42	35	101

12

Insertion sort

Simple sorting algorithm.

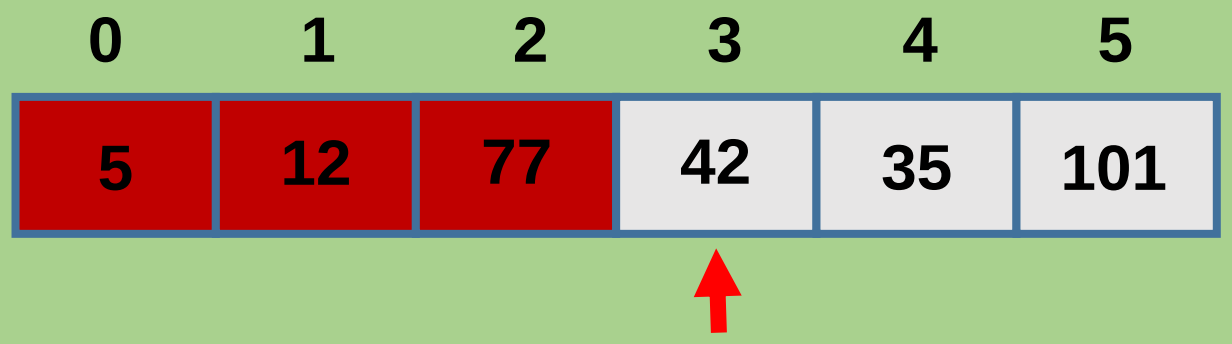
- ✓ n-1 passes over the array
- ✓ At the end of pass *i*, the elements that occupied A[0]…A[*i*] originally are still in those spots and in sorted order.

0	1	2	3	4	5
5	12	77	42	35	101

Insertion sort

Simple sorting algorithm.

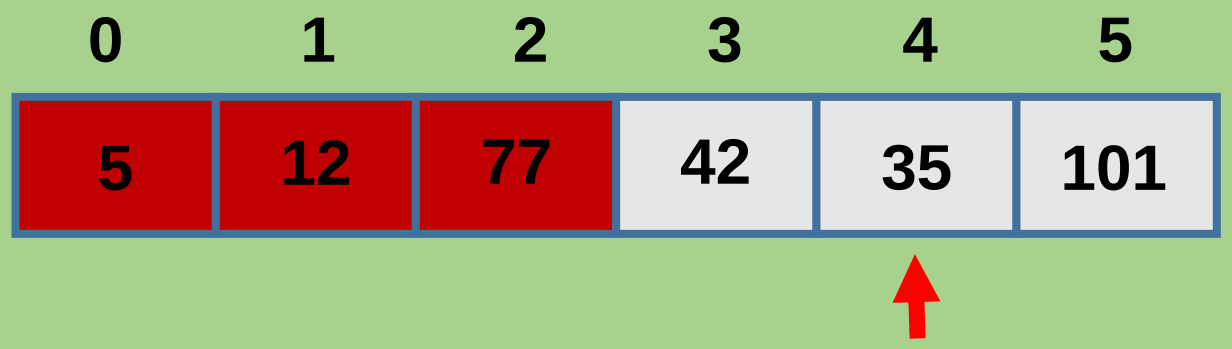
- ✓ n-1 passes over the array
- ✓ At the end of pass *i*, the elements that occupied A[0]…A[*i*] originally are still in those spots and in sorted order.



Insertion sort

Simple sorting algorithm.

- ✓ n-1 passes over the array
- ✓ At the end of pass *i*, the elements that occupied A[0]…A[*i*] originally are still in those spots and in sorted order.



Insertion sort

Simple sorting algorithm.

- ✓ n-1 passes over the array
- ✓ At the end of pass *i*, the elements that occupied A[0]…A[*i*] originally are still in those spots and in sorted order.

0	1	2	3	4	5
5	12	77	42	35	101

Insertion sort

Simple sorting algorithm.

- ✓ n-1 passes over the array
- ✓ At the end of pass *i*, the elements that occupied A[0]…A[*i*] originally are still in those spots and in sorted order.

0	1	2	3	4	5
5	12	77	42	35	101

Insertion sort

Simple sorting algorithm.

- ✓ n-1 passes over the array
- ✓ At the end of pass *i*, the elements that occupied A[0]…A[*i*] originally are still in those spots and in sorted order.

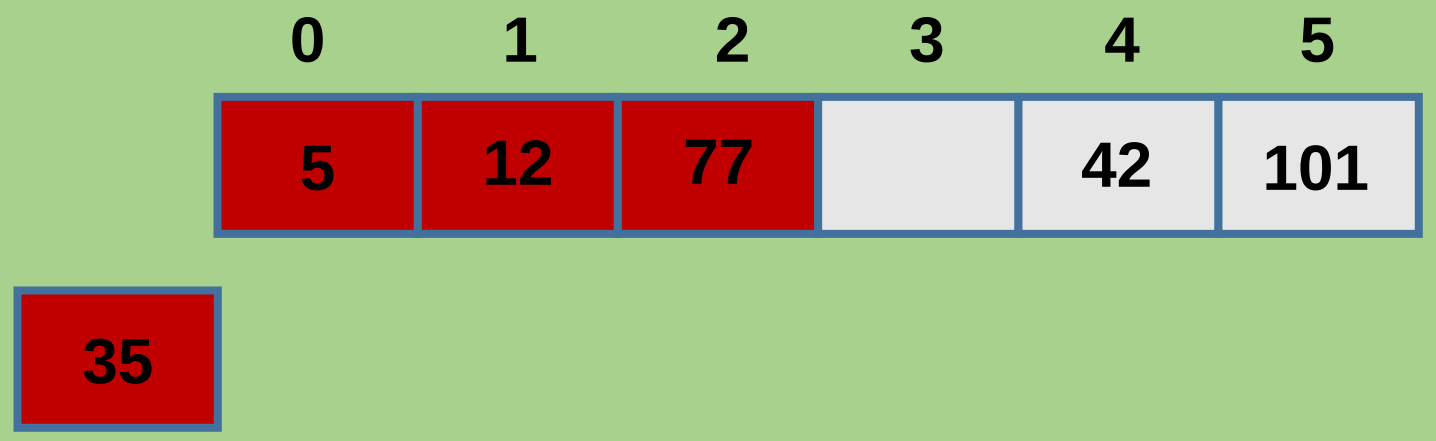
0	1	2	3	4	5
5	12	77	42		101

35

Insertion sort

Simple sorting algorithm.

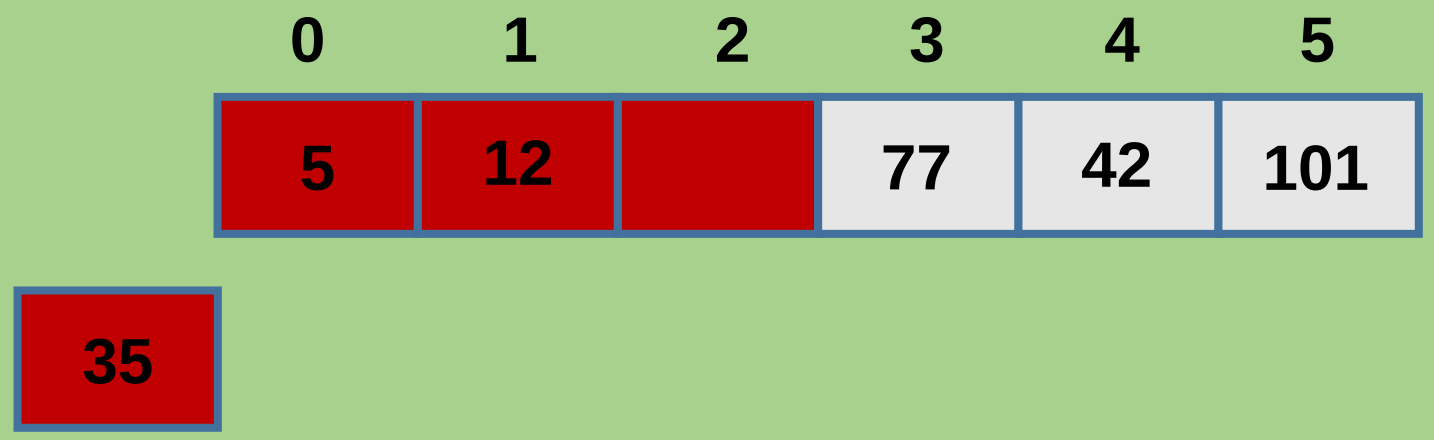
- ✓ n-1 passes over the array
- ✓ At the end of pass *i*, the elements that occupied A[0]…A[*i*] originally are still in those spots and in sorted order.



Insertion sort

Simple sorting algorithm.

- ✓ n-1 passes over the array
- ✓ At the end of pass *i*, the elements that occupied A[0]…A[*i*] originally are still in those spots and in sorted order.



Insertion sort

Simple sorting algorithm.

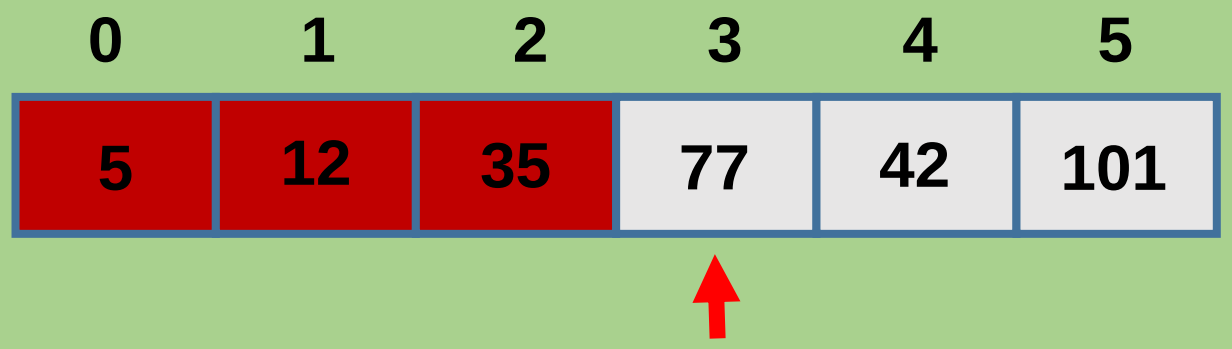
- ✓ n-1 passes over the array
- ✓ At the end of pass *i*, the elements that occupied A[0]…A[*i*] originally are still in those spots and in sorted order.

0	1	2	3	4	5
5	12	35	77	42	101

Insertion sort

Simple sorting algorithm.

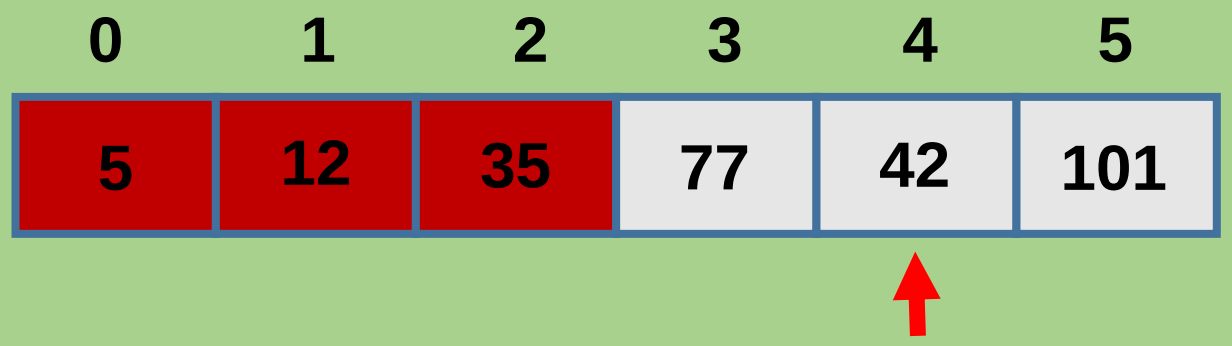
- ✓ n-1 passes over the array
- ✓ At the end of pass *i*, the elements that occupied A[0]…A[*i*] originally are still in those spots and in sorted order.



Insertion sort

Simple sorting algorithm.

- ✓ n-1 passes over the array
- ✓ At the end of pass *i*, the elements that occupied A[0]…A[*i*] originally are still in those spots and in sorted order.



Insertion sort

Simple sorting algorithm.

- ✓ n-1 passes over the array
- ✓ At the end of pass *i*, the elements that occupied A[0]…A[*i*] originally are still in those spots and in sorted order.

0	1	2	3	4	5
5	12	35	77	42	101

Insertion sort

Simple sorting algorithm.

- ✓ n-1 passes over the array
- ✓ At the end of pass *i*, the elements that occupied A[0]…A[*i*] originally are still in those spots and in sorted order.

0	1	2	3	4	5
5	12	35	77	42	101

Insertion sort

Simple sorting algorithm.

- ✓ n-1 passes over the array
- ✓ At the end of pass *i*, the elements that occupied A[0]…A[*i*] originally are still in those spots and in sorted order.

0	1	2	3	4	5
5	12	35	77		101

42

Insertion sort

Simple sorting algorithm.

- ✓ n-1 passes over the array
- ✓ At the end of pass *i*, the elements that occupied A[0]…A[*i*] originally are still in those spots and in sorted order.

0	1	2	3	4	5
5	12	35		77	101

42

Insertion sort

Simple sorting algorithm.

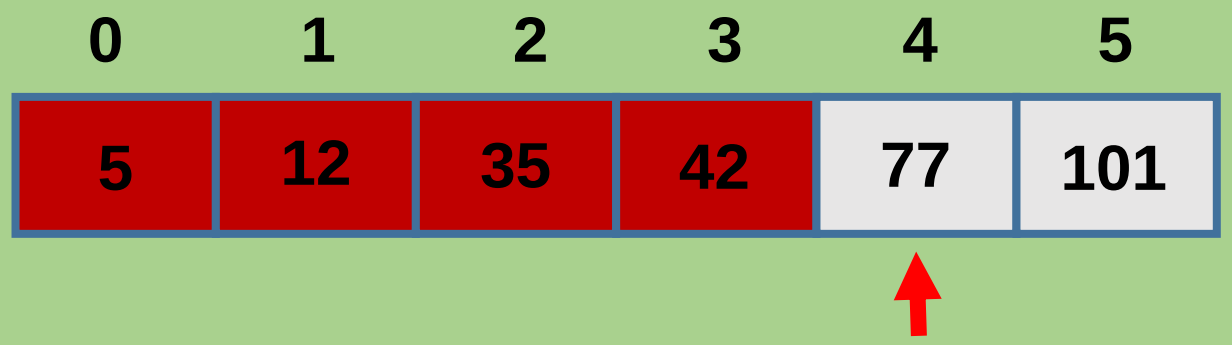
- ✓ n-1 passes over the array
- ✓ At the end of pass *i*, the elements that occupied A[0]…A[*i*] originally are still in those spots and in sorted order.

0	1	2	3	4	5
5	12	35	42	77	101

Insertion sort

Simple sorting algorithm.

- ✓ n-1 passes over the array
- ✓ At the end of pass *i*, the elements that occupied A[0]…A[*i*] originally are still in those spots and in sorted order.



Insertion sort

Simple sorting algorithm.

- ✓ n-1 passes over the array
- ✓ At the end of pass *i*, the elements that occupied A[0]…A[*i*] originally are still in those spots and in sorted order.

0	1	2	3	4	5
5	12	35	42	77	101

Insertion sort

Simple sorting algorithm.

- ✓ n-1 passes over the array
- ✓ At the end of pass *i*, the elements that occupied A[0]…A[*i*] originally are still in those spots and in sorted order.

0	1	2	3	4	5
5	12	35	42	77	101

Insertion sort

Simple sorting algorithm.

- ✓ n-1 passes over the array
- ✓ At the end of pass *i*, the elements that occupied A[0]…A[*i*] originally are still in those spots and in sorted order.

0	1	2	3	4	5
5	12	35	42	77	101

Insertion sort

Simple sorting algorithm.

- ✓ n-1 passes over the array
- ✓ At the end of pass *i*, the elements that occupied A[0]…A[*i*] originally are still in those spots and in sorted order.

0	1	2	3	4	5
5	12	35	42	77	101

Insertion sort Code

```
public static void insertionSort(int[] a) {
    for (int i = 1; i < a.length; i++) {
        int temp = a[i];

        // slide elements down to make room for a[i]
        int j = i;
        while (j > 0 && a[j - 1] > temp) {
            a[j] = a[j - 1];
            j--;
        }

        a[j] = temp;
    }
}
```

Insertion sort Runtime

worst case: reverse-ordered elements in array.

$$\sum_{i=1}^{n-1} i = 1+2+3+\ldots+(n-1) = \frac{(n-1)n}{2}$$

$$= \Theta(n^2)$$

best case: array is in sorted ascending order.

$$\sum_{i=1}^{n-1} 1 = n-1 = \Theta(n)$$

average case: each element is about halfway in order.

$$\sum_{i=1}^{n-1} \frac{i}{2} = \frac{1}{2}(1+2+3\ldots+(n-1)) = \frac{(n-1)n}{4}$$

$$= \Theta(n^2)$$