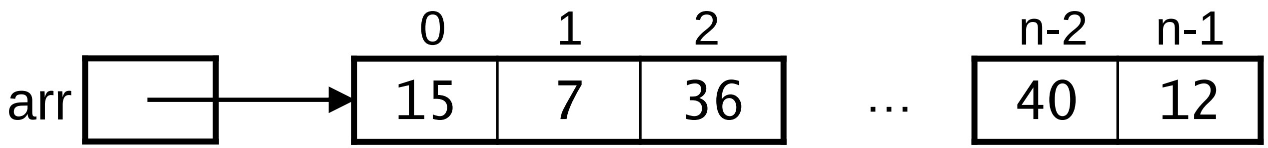
Sorting
and
Algorithm Analysis:
The Basics

Computer Science 112
Boston University

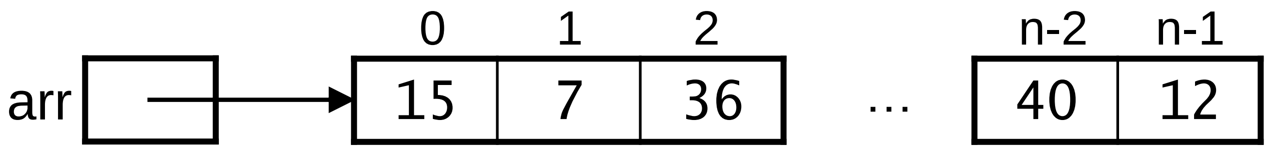
Christine Papadakis-Kanaris

Sorting an Array of Integers



- Ground rules:
  - sort the values in increasing order
  - sort "in place," using only a small amount of additional storage
- Terminology:
  - position: one of the memory locations in the array
  - element: one of the data items stored in the array
  - element i: the element at position i

Sorting an Array of Integers



- Ground rules:
  - sort the values in increasing order
  - sort “in place,” using only a small a

- Terminology:
  - position: one of the memory locatio
  - element: one of the data items stored in the array
  - element i: the element at position i

Which are?

number of

comparisons and *moves*

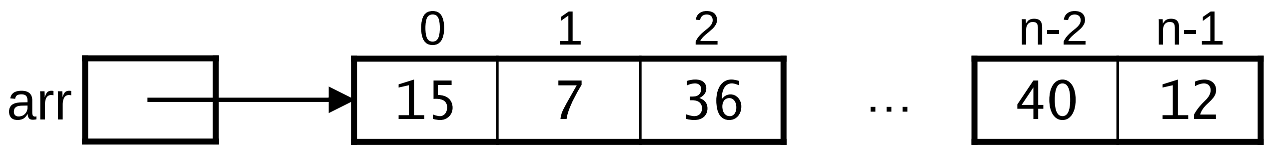
needed to sort the array.

- Goal: minimize the number of *operations* needed to sort the array.

```
comparison is applying a relational operation on two elements of the array.
example: arr[1] > arr[2]
```

```
move = copying an element from one position to another
example: arr[3] = arr[5];
```

Sorting an Array of Integers



- Ground rules:
  - sort the values in increasing order
  - sort “in place,” using only a small a
- Terminology:
  - position: one of the memory locatio
  - element: one of the data items stored in the array
  - element i: the element at position i
- Goal: minimize the number of *operations* needed to sort the array.

Which are?

number of *comparisons* and *moves* needed to sort the array.

```
comparison is applying a relational operation on two elements of the array.
example: arr[1] > arr[2]
```

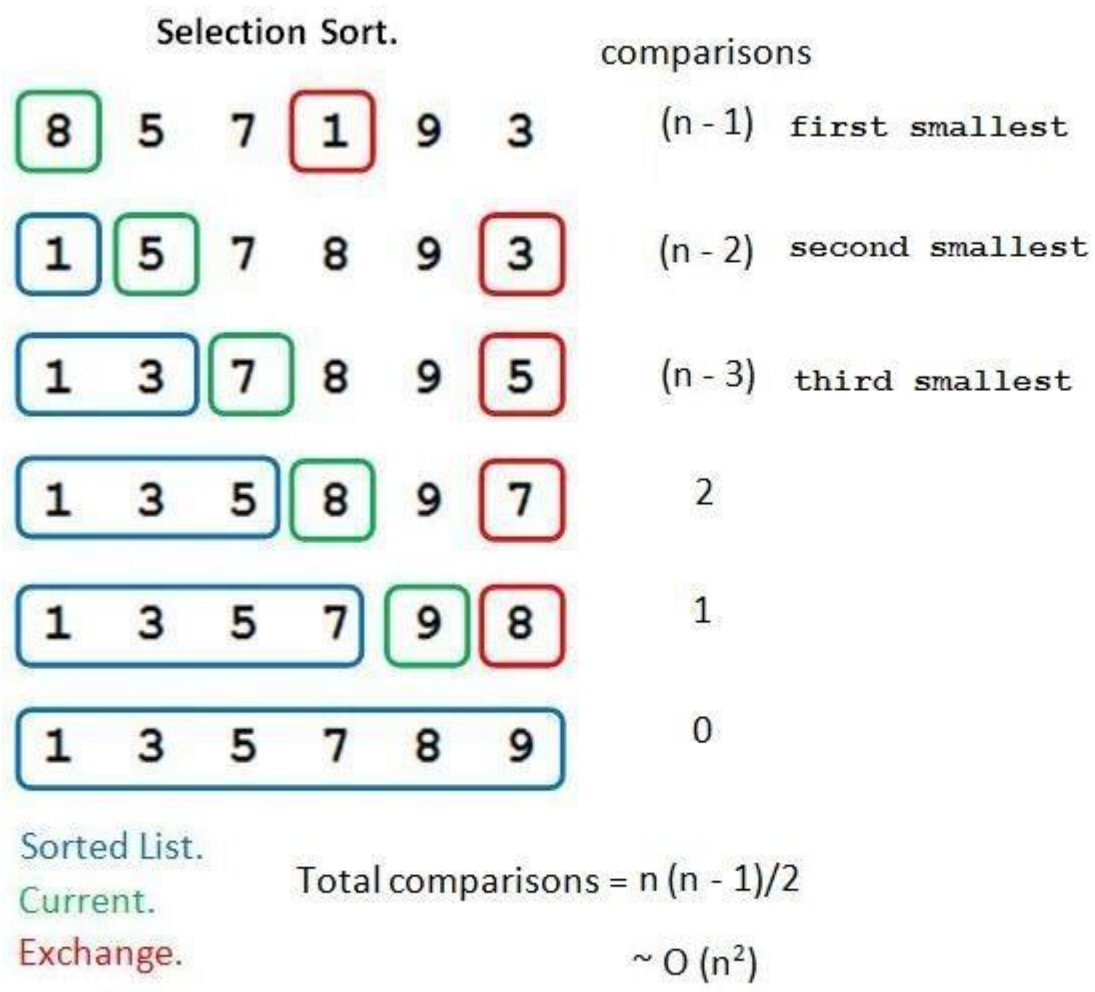
```
move = copying an element from one position to another
example: arr[3] = arr[5];
```

Defining a Class for our Sort Methods

```
public class Sort {
    public static void bubbleSort(int[] arr) {
        ...
    }
    public static void insertionSort(int[] arr) {
        ...
    }
    ...
}
```

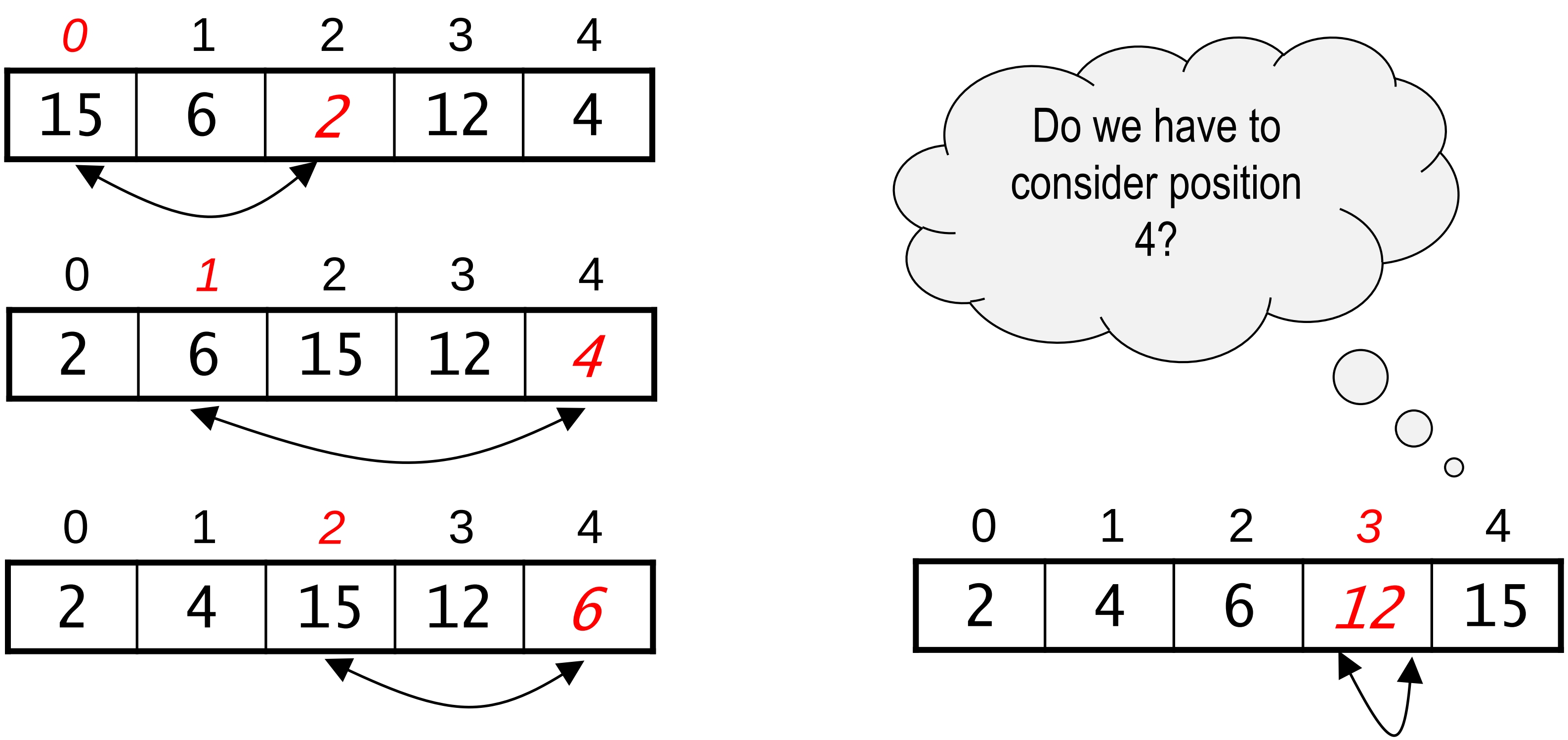
- Our `Sort` class is simply a collection of methods like Java's built-in `Math` class.
- Because we never create `Sort` objects, all of the methods in the class must be *static*.
    - outside the class, we invoke them using the class name: e.g., **`Sort.bubbleSort(arr)`**

Selection Sort



Selection Sort

- Basic idea:
  - consider the positions in the array from left to right
  - for each position, find the element that belongs there and put it in place by swapping it with the element that's currently there
- Example:



Selecting an Element

- When we consider position `i`, the elements in positions 0 through `i – 1` are already in their final positions.

example for `i` = 3:

0	1	2	3	4	5	6
2	4	7	21	25	10	17

- To select an element for position `i`:
  - consider elements `i`, `i+1,i+2,…,arr.length – 1`, and keep track of `indexMin`, the index of the smallest element seen thus far, example:

`indexMin: 3, 5`

0	1	2	3	4	5	6
2	4	7	21	25	*10*	17

  - when we finish this pass, `indexMin` is the index of the element that belongs in position `i`.
  - swap `arr[i]` and `arr[indexMin]`:

0	1	2	3	4	5	6
2	4	7	10	25	21	17

Algorithm for Selection Sort

```
• public static void selectionSort(int[] arr) {
      // scan every position from current to length-1 {
          // find the index containing the smallest element

          // swap the current element (i.e. element at i)
          // with the element at the index containing the
          // smallest element
      }
  }
```

Algorithm for Selection Sort

```
public static void selectionSort(int[] arr) {
    for (int i = 0; i < arr.length - 1; i++) {
        // find the index containing the smallest element

        // swap the cur                            at i)
        // with the el                          g the
        // smallest
    }
}
```

… from position i to
the last element of
the array!

Algorithm for Selection Sort

```
public static void selectionSort(int[] arr) {
    for (int i = 0; i < arr.length - 1; i++) {
        int j = indexSmallest( arr, i );

        // swap the cur                          at i)
        // with the el                         g the
        // smallest
    }
}
```

… from position i to the last element of the array!

Implementation of Selection Sort

- Use a *helper* method to find the index of the smallest element:

```
private static int indexSmallest(int[] arr, int start) {
    int indexMin = start;

    for (int i = start + 1; i < arr.length; i++) {
        if (arr[i] < arr[indexMin]) {
            indexMin = i;
        }
    }

    return indexMin;
}
```

Algorithm for Selection Sort

- ```
  public static void selectionSort(int[] arr) {
      for (int i = 0; i < arr.length - 1; i++) {
          int j = indexSmallest( arr, i );

          // swap the current element (i.e. element at i)
          // with the element at the index containing the
          // smallest element
      }
  }
  ```
  ```

Algorithm for Selection Sort

- ```
  public static void selectionSort(int[] arr) {
      for (int i = 0; i < arr.length - 1; i++) {
          int j = indexSmallest( arr, i );

          swap(arr, i, j);


      }
  }
  ```
  ```

A Method for Swapping Elements
within an array

- A private helper method used by several of the algorithms:

```
private static void swap(int[] arr, int a, int b) {
    int temp = arr[a];
    arr[a] = arr[b];
    arr[b] = temp;
}
```

- For example:

```
int[] arr = {15, 7, 3, 6, 12};
swap(arr, 0, 1);
System.out.println(Arrays.toString(arr));
```

A Method for Swapping Elements

within an array

- A private helper method used by several of the algorithms:

```
private static void swap(int[] arr, int a, int b) {
    int temp = arr[a];
    arr[a] = arr[b];
    arr[b] = temp;
}
```

Where ***a*** an ***b*** are positions (i.e. indices) into the array.

- For example:

```
int[] arr = {15, 7, 3, 6, 12};
swap(arr, 0, 1);
System.out.println(Arrays.toString(arr));
```

A Method for Swapping Elements

within an array

- A private helper method used by several of the algorithms:

```
private static void swap(int[] arr, int a, int b) {
    int temp = arr[a];
    arr[a] = arr[b];
    arr[b] = temp;
}
```

- For example:

```
int[] arr = {15, 7, 3, 6, 12};
swap(arr, 0, 1);
System.out.println(Arrays.toString(arr));
```

output:

```
[7, 15, 3, 6, 12]
```

Another method for Swapping Variables?

```
private static void swap( int a, int b ) {

   int tmp = a;

   a = b;
   b = tmp;

}
```

Where ***a*** an ***b*** are variables whose values we want to swap.

Will this method swap the values of variables *a* and *b*? Why or why not? For example:

```
   int x = 5, y = 10;
   System.out.println(x + “ “ + y); // What would this print?
   swap(x,y);
   System.out.println(x + “ “ + y); // What would this print?
```

And this method for Swapping Variables?

```
public static void swap( Integer a, Integer b ) {

    Integer tmp = new Integer(a);

    a = b;
    b = tmp;
}
```

Where ***a*** an ***b*** are references to integer objects whose values we want to swap.

Will this method swap the values of variables *a* and *b*? Why or why not? For example:

```
Integer x = new Integer(5), y = new Integer(10);
System.out.println(x + " " + y); // What would this print?
swap(x,y);
System.out.println(x + " " + y); // What would this print?
```

Algorithm for Selection Sort

```
public static void selectionSort(int[] arr) {
    for (int i = 0; i < arr.length - 1; i++) {
        int j = indexSmallest( arr, i );

        swap(arr, i, j);


    }
}
```

Note that the swap is being performed even if the minimum index returned is equal to i. Consider why….

Algorithm for Selection Sort

```
public static void selectionSort(int[] arr) {
    for (int i = 0; i < arr.length - 1; i++) {
        int j = indexSmallest( arr, i );

        if (i != j)
            swap(arr, i, j);

    }
}
```

You would have to perform the comparison each time!

Analysis of Selection Sort

- Use a helper method to find the index of the smallest element:

```
private static int indexSmallest(int[] arr, int start) {
    int indexMin = start;

    for (int i = start + 1; i < arr.length; i++) {
        if (arr[i] < arr[indexMin]) {
            indexMin = i;
        }
    }

    return indexMin;
}
```

- The actual sort method is very simple:

```
public static void selectionSort(int[] arr) {
    for (int i = 0; i < arr.length - 1; i++) {
        int j = indexSmallest(arr, i);
        swap(arr, i, j);
    }
}
```

Time Analysis

- The *time efficiency* or *time complexity* of an algorithm is some measure of the number of "operations" that it performs.
  - for sorting algorithms, we'll focus on two types of operations: *comparisons* and *moves*
- The number of operations that an algorithm performs typically depends on the size, `n`, of its input.
  - for sorting algorithms, n is the # of elements in the array
  - `C(n)` = number of comparisons
  - `M(n)` = number of moves
- To express the time complexity of an algorithm, we'll express the number of operations performed as a function of `n`.
  - examples: $C(n) = n^2 + 3n$

    $M(n) = 2n^2 - 1$

Counting Comparisons by Selection Sort

```
private static int indexSmallest(int[] arr, int start){
    int indexMin = start;

    for (int i = start + 1; i < arr.length; i++) {
        if (arr[i] < arr[indexMin]) {
            indexMin = i;
        }
    }

    return indexMin;
}
public static void selectionSort(int[] arr) {
    for (int i = 0; i < arr.length - 1; i++) {    // n - 1 iterations
        int j = indexSmallest(arr, i);

        swap(arr, i, j);
    }
}
```

- To sort n elements, selection sort performs n – 1 iterations:

Counting Comparisons by Selection Sort

```
private static int indexSmallest(int[] arr, int start){
    int indexMin = start;

    for (int i = start + 1; i < arr.length; i++) {
        if (arr[i] < arr[indexMin]) {
            indexMin = i;
        }
    }

    return indexMin;
}
public static void selectionSort(int[] arr) {
    for (int i = 0; i < arr.length - 1; i++) {    // n - 1 iterations
        int j = indexSmallest(arr, i);   // each iteration performs
                                         // one pass starting at i
        swap(arr, i, j);
    }
}
```

- To sort n elements, selection sort performs n – 1 iterations:

Counting Comparisons by Selection Sort

```
private static int indexSmallest(int[] arr, int start){
    int indexMin = start;

    for (int i = start + 1; i < arr.length; i++) {
        if (arr[i] < arr[indexMin]) {
            indexMin = i;
        }
    }

    return indexMin;
}
public static void selectionSort(int[] arr) {
    for (int i = 0; i < arr.length - 1; i++) {
        int j = indexSmallest(arr, i);

        swap(arr, i, j);
    }
}
```

- To sort n elements, selection sort performs n – 1 passes:

  on 1st pass, it performs n - 1 comparisons to find `indexSmallest`

  on 2nd pass, it performs n - 2 comparisons

  ...

  on the (n–1)st pass, it performs 1 comparison

- Adding them up: C(n) = 1 + 2 + … + (n - 2) + (n - 1)

Counting Comparisons by Selection Sort (cont.)

- The resulting formula for `C(n)` is the sum of an arithmetic sequence:

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

Counting Comparisons by Selection Sort (cont.)

- The resulting formula for `C(n)` is the sum of an arithmetic sequence:

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

- Closed Formula for the sum of this type of arithmetic sequence:

$$\boxed{\sum_{i=1}^{m} i = \frac{m(m+1)}{2}}$$

Substitute (n-1) for **m**

Counting Comparisons by Selection Sort (cont.)

- The resulting formula for `c(n)` is the sum of an arithmetic sequence:

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

- Closed Formula for the sum of this type of arithmetic sequence:

$$\sum_{i=1}^{m} i = \frac{m(m+1)}{2}$$

- Thus, we can re-write our expressio

factor out (n-1)

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

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

Counting Comparisons by Selection Sort (cont.)

- The resulting formula for C(n) is the sum of an arithmetic sequence:

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

- Closed Formula for the sum of this type of arithmetic sequence:

$$\sum_{i=1}^{m} i = \frac{m(m+1)}{2}$$

- Thus, we can re-write our expression for C(n) as follows:

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

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

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

$$C(n) = n^2/2 - n/2$$

Big-O Notation

- We specify the largest term using big-O notation.
  - e.g., we say that $C(n) = n^2/2 - n/2$ is $\boxed{O(n^2)}$

As **n** increases and approaches *infinity*, the solution grows in proportion to **n^2**

Big-*O* Time Analysis of Selection Sort

- Comparisons: we showed that $C(n) = n^2/2 - n/2$
  - selection sort performs $O(n^2)$ comparisons

- Moves: after each of the n-1 passes, the algorithm does one swap.

```
public static void selectionSort(int[] arr) {
    for (int i = 0; i < arr.length - 1; i++) {
        int j = indexSmallest(arr, i);
        swap(arr, i, j);
    }
}
```

Big-*O* Time Analysis of Selection Sort

- Comparisons: we showed that $C(n) = n^2/2 - n/2$
  - selection sort performs $O(n^2)$ comparisons
- Moves: after each of the $n-1$ passes, the algorithm does one swap.
  - $n-1$ swaps, 3 moves per swap
  - $M(n) = 3(n-1) = 3n-3$
  - selection sort performs $O(n)$ moves.

Big-O Time Analysis of Selection Sort

- Comparisons: we showed that $C(n) = n^2/2 - n/2$
  - selection sort performs $O(n^2)$ comparisons

- Moves: after each of the $n-1$ passes, the algorithm does one swap.
  - $n-1$ swaps, 3 moves per swap
  - $M(n) = 3(n-1) = 3n-3$
  - selection sort performs $O(n)$ moves.

- Running time (i.e., total operations): $O(n^2)$
  - $C(n) = O(n^2)$
  - $M(n) = O(n)$
  - therefore, the largest term of $C(n) + M(n)$ is $O(n^2)$

- Selection sort is a *quadratic-time* or $O(n^2)$ algorithm.

Big-*O* Notation

- We specify the largest term using big-*O* notation.
  - e.g., we say that $c(n) = n^2/2 - n/2$ is $O(n^2)$

- Common classes of algorithms:

slower

name	example expressions	big-O notation
constant time	$1, 7, 10$	$O(1)$
logarithmic time	$3\log_{10} n, \log_2 n + 5$	$O(\log n)$
linear time	$5n, 10n - 2\log_2 n$	$O(n)$
$n \log n$ time	$4n\log_2 n, n\log_2 n + n$	$O(n \log n)$
quadratic time	$2n^2 + 3n, n^2 - 1$	$O(n^2)$
exponential time	$2^n, 5e^n + 2n^2$	$O(c^n)$

Big-*O* Notation

- We specify the largest term using big-*O* notation.
  - e.g., we say that $c(n) = n^2/2 - n/2$ is $O(n^2)$

- Common classes of algorithms:

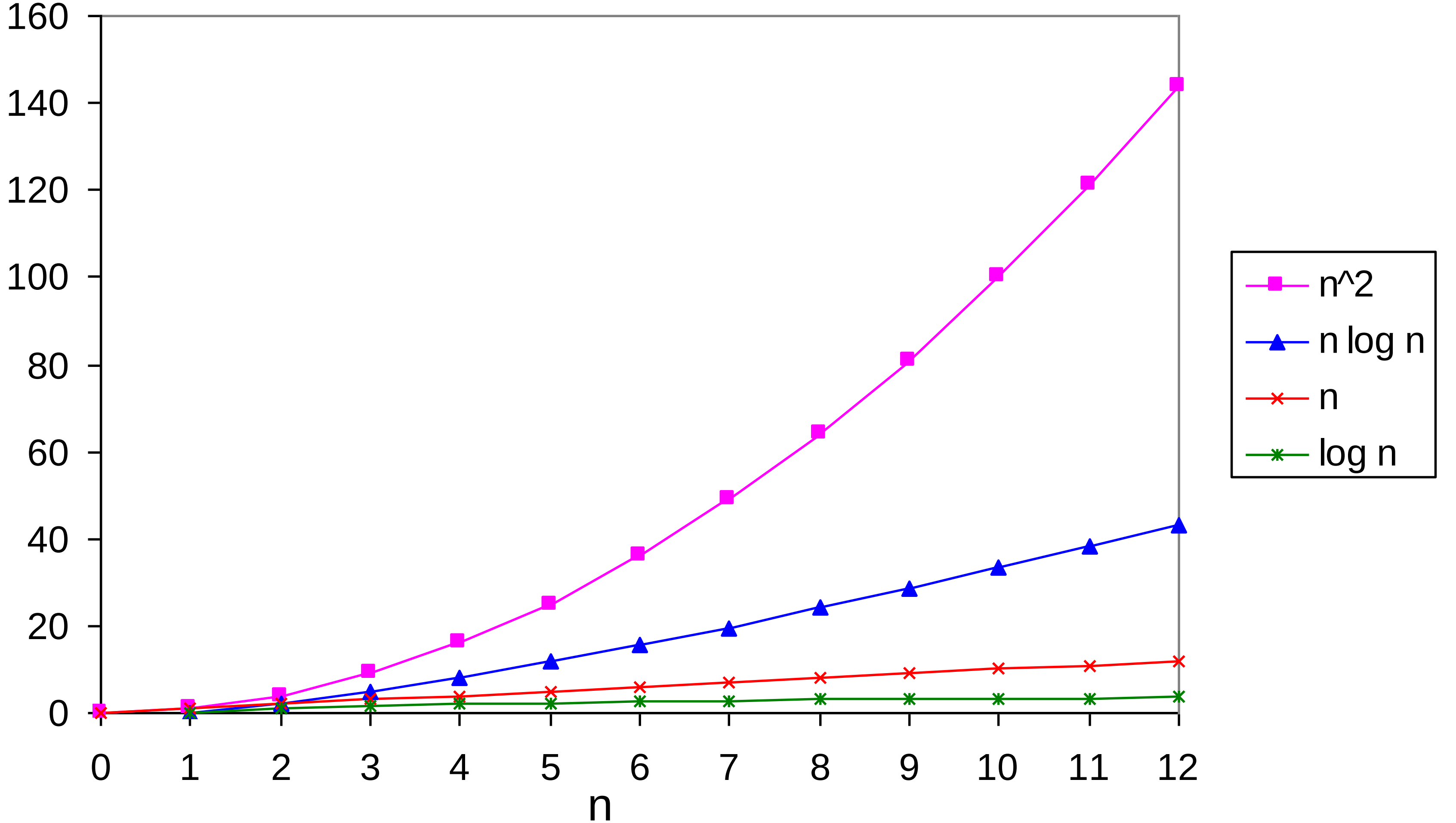
name	example expressions	big-O notation
constant time	$1, 7, 10$	$O(1)$
logarithmic time	$3\log_{10} n,\ \log_2 n + 5$	$O(\log n)$
linear time	$5n,\ 10n - 2\log_2 n$	$O(n)$
$n\log n$ time	$4n\log_2 n,\ n\log_2 n + n$	$O(n\log n)$
quadratic time	$2n^2 + 3n,\ n^2 - 1$	$O(n^2)$
exponential time	$2^n,\ 5e^n + 2n^2$	$O(c^n)$

slower

- For large inputs, efficiency matters more than CPU speed.
  - e.g., an $O(\log n)$ algorithm on a slow machine will outperform an $O(n)$ algorithm on a fast machine

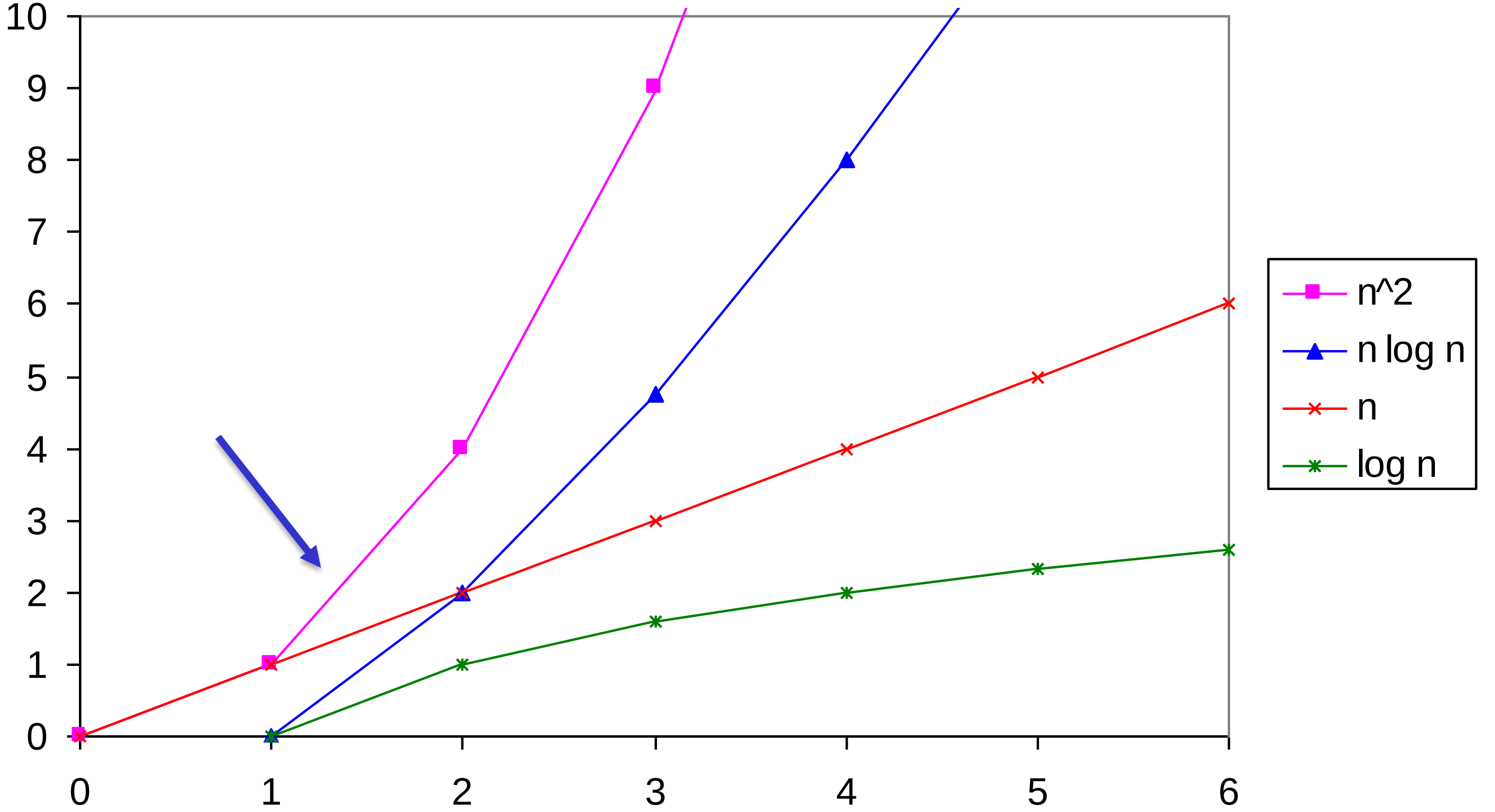
Ordering of Functions

- We can see below that: n^2 grows faster than $n\log_2 n$
  $n\log_2 n$ grows faster than n
  n grows faster than $\log_2 n$

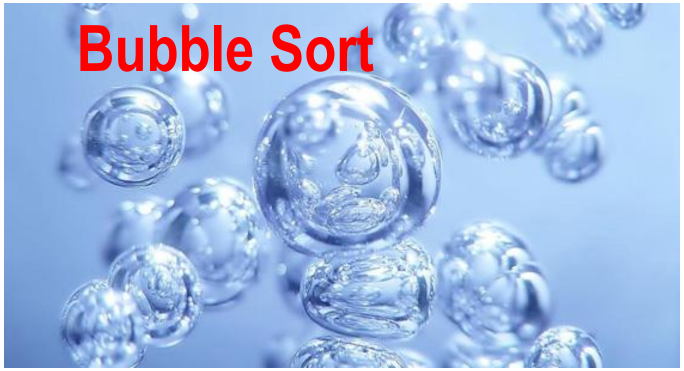


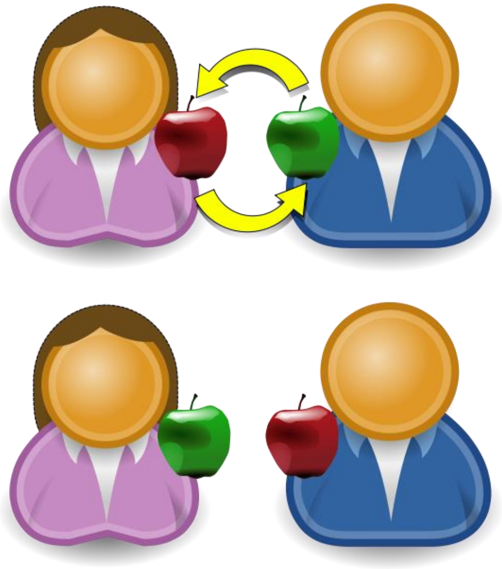
Ordering of Functions (cont.)

- Zooming in, we see that: $n^2 >= n$ for all $n >= 1$
  $n \log_2 n >= n$ for all $n >= 2$
  $n > \log_2 n$ for all $n >= 1$



Exchange based sorting algorithms





Selection Sort

SWAP

Sorting by Exchange:
Bubble Sort

- Perform a sequence of passes through the array.
- On each pass: proceed from left to right, swapping adjacent elements if they are out of order.
- Larger elements *bubble up* to the end of the array.
- At the end of the kth pass, the k rightmost elements are in their final positions, so we don't need to consider them in subsequent passes.
- Example:

	0	1	2	3
	28	24	27	18
after the first pass:	24	27	18	*28*
after the second:	24	18	*27*	*28*
after the third:	18	*24*	*27*	*28*

Implementation of Bubble Sort

```
public class Sort {
    ...
    public static void bubbleSort(int[] arr) {
        for (int i = arr.length - 1; i > 0; i--) {
            for (int j = 0; j < i; j++) {
                if (arr[j] > arr[j+1])
                    swap(arr, j, j+1);
            }
        }
    }
}
```

- One for-loop nested in another:
    - the inner loop performs a single pass
    - the outer loop governs the number of passes, <u>and</u> the *ending point* of each pass

Time Analysis of Bubble Sort

- Comparisons: the kth pass performs $\underline{n - k}$ comparisons, so we get $C(n) = \sum_{i=1}^{n-1} i = n^2/2 - n/2 = O(n^2)$
- Moves: depends on the contents of the array
  - in the worst case: the array is in reverse order, and every comparison leads to a swap (3 moves)
    - $M(n) = 3C(n) = O(n^2)$
  - in the best case: the array is already sorted, and no moves are needed
- Running time: $O(n^2)$
  - $C(n)$ is always $O(n^2)$, $M(n)$ is never worse than $O(n^2)$
  - therefore, the largest term of $C(n) + M(n)$ is $O(n^2)$
- Bubble sort is a quadratic-time or $O(n^2)$ algorithm.
  - same run-time efficiency as selection sort!

Time Analysis of Bubble Sort

- Comparisons: the kth pass performs $\underline{n - k}$ comparisons,

  so we get $C(n) = \sum_{i=1}^{n-1} i = n^2/2 - n/2 = O(n^2)$

- Moves: depends on the contents of the array
  - in the worst case: the array is in reverse order, and every comparison leads to a swap (3
    - $M(n) = 3C(n) = O(n^2)$
  - in the best case: the array is a moves are needed

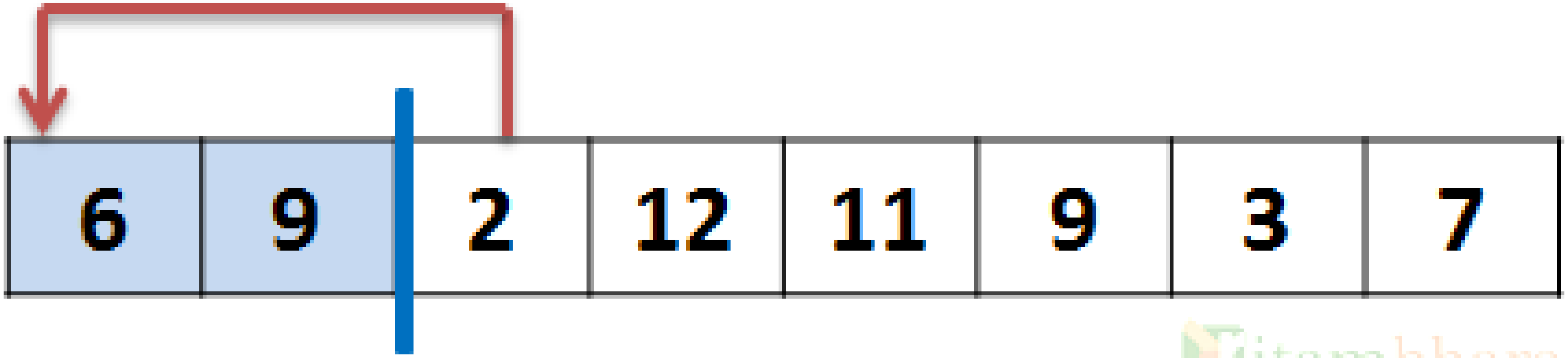
- Running time: $O(n^2)$
  - $C(n)$ is always $O(n^2)$, $M(n)$ is
  - therefore, the largest term of

Unless we optimize the algorithm to determine when the array is sorted and stop the loop from making any additional passes:

$O(n)$ *in best case!*

- Bubble sort is a quadratic-time or $O(n^2)$ algorithm.
  - same run-time efficiency as selection sort!

Sorting by Insertion



Insertion Sort

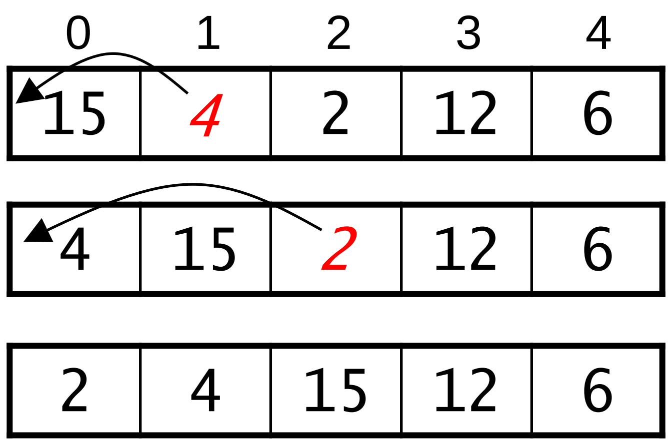
- Basic idea:
  - going from left to right, “insert” each element into its proper place with respect to the elements to its left, “sliding over” other elements to make room.
- Example:

0	1	2	3	4
15	4	2	12	6

Insertion Sort

- Basic idea:
  - going from left to right, “insert” each element into its proper place with respect to the elements to its left, “sliding over” other elements to make room.

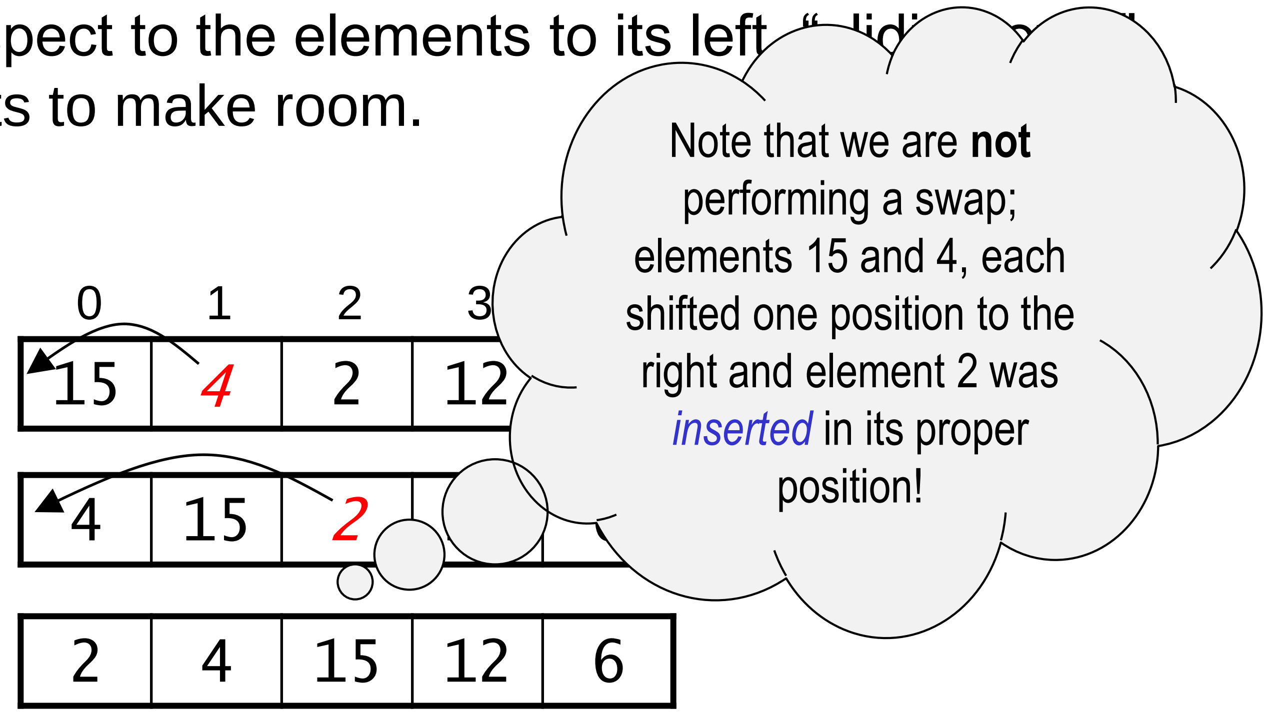
- Example:



Insertion Sort

- Basic idea:
  - going from left to right, “insert” each element into its proper place with respect to the elements to its left, “sli... other elements to make room.

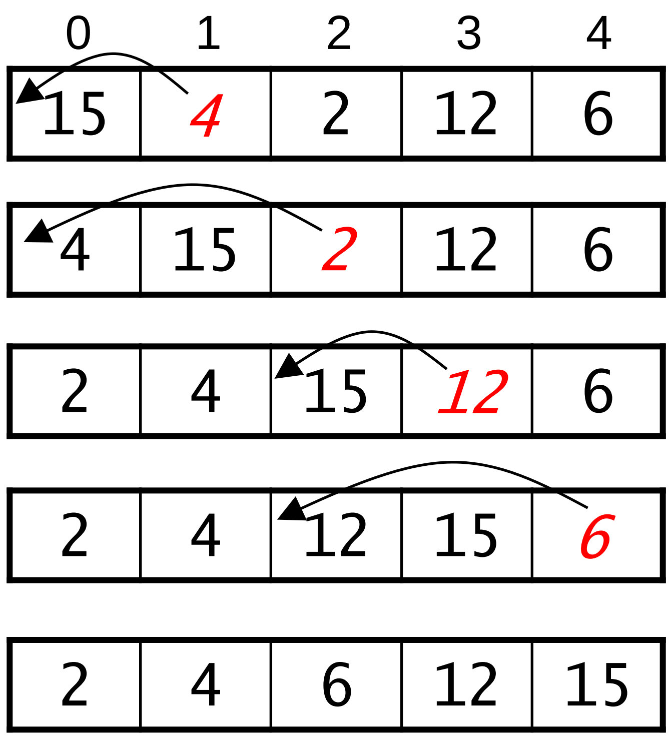
- Example:



Insertion Sort

- Basic idea:
  - going from left to right, “insert” each element into its proper place with respect to the elements to its left, “sliding over” other elements to make room.

- Example:



Comparing Selection and Insertion Strategies

- In selection sort, we start with the *positions* in the array and *select* the correct elements to fill them.
- In insertion sort, we start with the *elements* and determine where to *insert* them in the array.
- Here's an example that illustrates the difference:

0	1	2	3	4	5	6
18	12	15	9	25	2	17

- Sorting by selection:
  - consider position 0: find the element (2) that belongs there
  - consider position 1: find the element (9) that belongs there
  - …
- Sorting by insertion:
  - consider the 12: determine where to insert it
  - consider the 15; determine where to insert it
  - …

Comparing Selection and Insertion Strategies

- In selection sort, we start with the *positions* in the array and *select* the correct elements to fill them.
- In insertion sort, we start with the *elements* and determine where to *insert* them in the array.
- Here's an example that illustrates the difference:

0	1	2	3	4	5	6
18	12	15	9	25	2	17

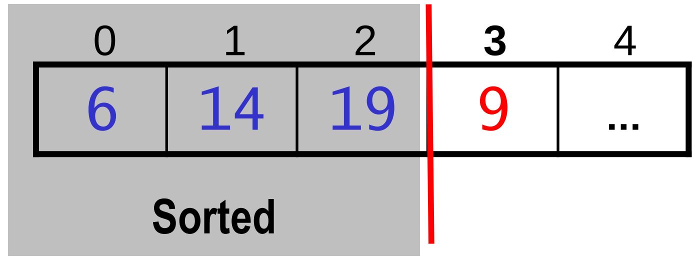
- Sorting by selection:
  - consider position 0: find the element (2) that belongs there
  - consider position 1: find the element (9) that belongs there
  - …
- Sorting by insertion:
  - consider the 12: determine where to insert it
  - consider the 15; determine where to insert it
  - …

Inserting an Element:

following the algorithm

- When we consider element `i`, elements `0` through `i – 1` are already sorted with respect to each other.

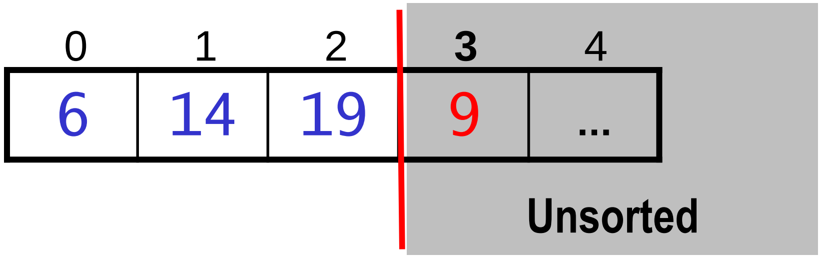
example for `i = 3`:



Inserting an Element:
following the algorithm

- When we consider element `i`, elements `0` through `i – 1` are already sorted with respect to each other.

example for `i = 3`:



Inserting an Element:

following the algorithm

- When we consider element `i`, elements 0 through `i – 1` are already sorted with respect to each other.

example for `i = 3`:

0	1	2	3	4
6	14	19	9	...

- To insert element `i`:
  - make a copy of element `i`, storing it in the variable `toInsert`:

`toInsert` 9

0	1	2	3
6	14	19	9

  - consider elements `i-1`, `i-2`, …
    - if an element `> toInsert`, slide it over to the right
    - stop at the first element `<= toInsert`

`toInsert` 9

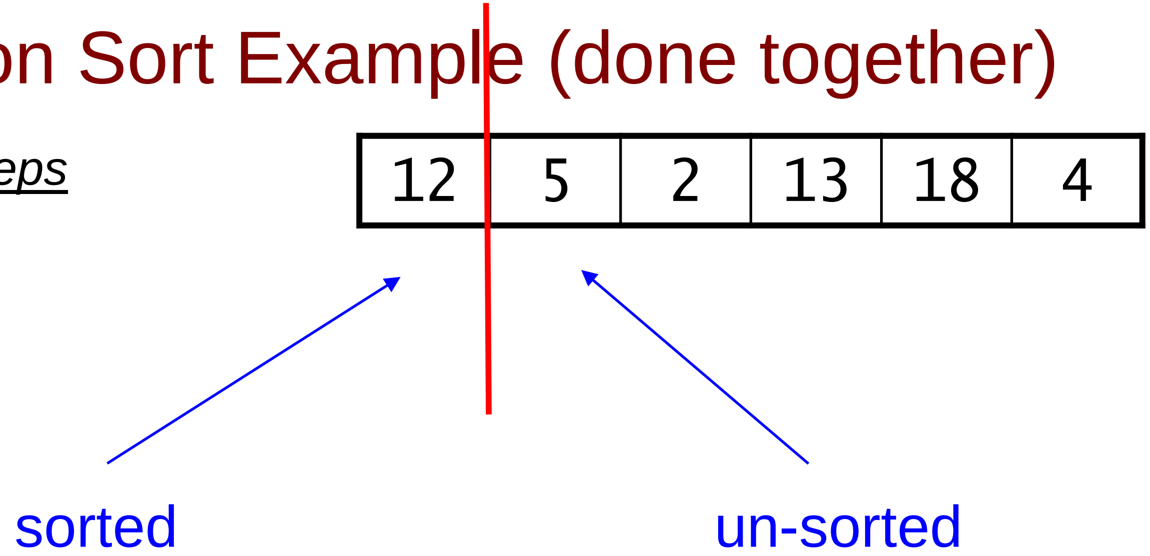
0	1	2	3
6		14	19

  - copy `toInsert` into the resulting “hole”:

0	1	2	3
6	9	14	19

Insertion Sort Example (done together)

description of steps



Insertion Sort Example (done together)

description of steps

12	5	2	13	18	4

1. copy the 5

5

12	5	2	13	18	4

Note by copying this value somewhere, we have opened up a slot that elements in the sorted region can *slide* into.

Insertion Sort Example (done together)

description of steps							
		12	5	2	13	18	4
1. copy the 5; shift the 12 to make room	5		12	2	13	18	4
2. insert the 5		5	12	2	13	18	4
3. copy the 2; shift the 12 and the 5 to make room	2		5	12	13	18	4
4. insert the 2		2	5	12	13	18	4
5. 13 > 12, so no need to insert it; similarly, 18 > 13		2	5	12	13	18	4
6. copy the 4; shift 18,13, 12, and 5 to make room	4	2		5	12	13	18
7. insert the 4		2	4	5	12	13	18

Implementation of Insertion Sort

```
public class Sort {
  ...
  public static void insertionSort(int[] arr) {
      for (int i = 1; i < arr.length; i++) {
          if (arr[i] < arr[i-1]) {
              int toInsert = arr[i];

              int j = i;
              do {
                  arr[j] = arr[j-1];
                  j = j - 1;
              } while (j > 0  &&  toInsert < arr[j-1]);

              arr[j] = toInsert;
          }
      }
  }
}
```

Implementation of Insertion Sort

```
public class Sort {
   ...
   public static void insertionSort(int[] arr) {
      for (int i = 1; i < arr.length; i++) {
         if (arr[i] < arr[i-1]) {
            int toInsert = arr[i];

            int j = i;
            do {
               arr[j] = arr
               j = j - 1;
            } while (j > 0  &

            arr[j] = toInsert;
         }
      }
   }
}
```

Note that control variable `i` represents the 1st element in the *unsorted* region. All the elements to the left are sorted.

Implementation of Insertion Sort

```
public class Sort {
  ...
  public static void insertionSort(int[] arr) {
      for (int i = 1; i < arr.length; i++) {
          if (arr[i] < arr[i-1]) {
              int toInsert = arr[i];

              int j = i;
              do {
                  arr[j] = arr[j-1];
                  j = j - 1;
              } while (j > 0  &&  toInsert < arr[j-1]);

              arr[j] = toInsert;
          }
      }
  }
}
```

Implementation of Insertion Sort

```
public class Sort {
   ...
   public static void insertionSort(int[] arr) {
      for (int i = 1; i < arr.length; i++) {
         if (arr[i] < arr[i-1]) {
            int toInsert   arr[i];

            int j = i;
            do {
               arr[j] = arr[j-
               j = j - 1;
            } while (j > 0  &

            arr[j] = toInsert
         }
      }
   }
}
```

If the element at position `i` is *smaller* than the element at position `i-1`, find the proper place in the *sorted* region to insert the element.

Implementation of Insertion Sort

```
public class Sort {
   ...
   public static void insertionSort(int[] arr) {
         for (int i = 1; i < arr.length; i++) {
               if (arr[i] < arr[i-1]) {
                     int toInsert = arr[i]; // save the element

                     int j = i;
                     do {
                           arr[j] = arr[j-1];
                           j = j - 1;
                     } while (j > 0  &&  toInsert < arr[j-1]);

                     arr[j] = toInsert;
               }
         }
   }
}
```

Implementation of Insertion Sort

```
public class Sort {
  ...
  public static void insertionSo
      for (int i = 1; i < arr.l
          if (arr[i] < arr[i-1])
              int toInsert = arr

              int j = i;
              do {
                  arr[j] = arr[j-1];
                  j = j - 1;
              } while (j > 0  &&  toInsert < arr[j-1]);

              arr[j] = toInsert;
          }
      }
  }
}
```

Loop through the *sorted* region to find the correct position to insert the element saved in variable *toInsert*.

Implementation of Insertion Sort

```
public class Sort {
   ...
   public static void insertionSo
      for (int i = 1; i < arr.l
         if (arr[i] < arr[i-1])
            int toInsert = arr

            int j = i;
            do {
               arr[j] = arr[j-1];
               j = j - 1;
            } while (j > 0  &&  toInsert < arr[j-1]);

            arr[j] = toInsert;
         }
      }
   }
}
```

Loop through the *sorted* region to find the correct position to insert the element saved in variable *toInsert*.

while there are more elements to check in the sorted region!

Implementation of Insertion Sort

```
public class Sort {
   ...
   public static void insertionSo
         for (int i = 1; i < arr.l
               if (arr[i] < arr[i-1])
                     int toInsert = arr

                     int j = i;
                     do {
                           arr[j] = arr[j-1];
                           j = j - 1;
                     } while (j > 0  &&  toInsert < arr[j-1]);

                     arr[j] = toInsert;
               }
         }
   }
}
```

Loop through the *sorted* region to find the correct position to insert the element saved in variable *toInsert*.

… and the element to insert is less than the elements being considered in the sorted region

Implementation of Insertion Sort

```
public class Sort {
  ...
  public static void insertionSort(int[] arr) {
      for (int i = 1; i < arr.length; i++) {
          if (arr[i] < arr[i-1]) {
              int toInsert = arr[i];

              int j = i;
              do {
                  arr[j] = arr[j-1];
                  j = j - 1;
              } while (j > 0  &&  toInsert < arr[j-1]);

              arr[j] = toInsert;
          }
      }
  }
}
```

Time Analysis of Insertion Sort

- The number of operations depends on the contents of the array.
- *best case:* array is sorted
  - thus, we never execute the do-while loop
  - each element is only compared to the element to its left
  - $C(n) = n - 1 = O(n)$, $M(n) = 0$, running time $= O(n)$

also true if array is almost sorted

Time Analysis of Insertion Sort

- The number of operations depends on the contents of the array.
- *best case:* array is sorted
  - thus, we never execute the do-while loop
  - each element is only compared to the element to its left
  - `C(n)` = `n – 1` = O`(n)`, `M(n)` = `0`, running time = O`(n)`

also true if array is almost sorted

- *worst case:* array is in reverse order
  - each element is compared to *all* of the elements to its left:

    `arr[1]` is compared to 1 element (`arr[0]`)
    `arr[2]` is compared to 2 elements (`arr[0]` and `arr[1]`)
    ...
    `arr[n-1]` is compared to `n-1` elements
  - `C(n)` = `1` + `2` + … + `(n – 1)` = $O(n^2)$ as seen in selection sort
  - similarly, `M(n)` = $O(n^2)$, running time = $O(n^2)$

- *average case:* elements are randomly arranged
  - each element is compared to *half* of the elements to its left
  - still get `C(n)` = `M(n)` = $O(n^2)$, running time = $O(n^2)$