

Revised 27/07/03

Sorting - Insertion Sort

Cmput 115 - Lecture 10
Department of Computing Science
University of Alberta
©Duane Szafron 2000

Some code in this lecture is based on code from the book:
Java Structures by Duane A. Bailey or the companion structure package

About This Lecture


- In this lecture we will learn about a sorting algorithm called the Insertion Sort.
- We will study its implementation and its time and space complexity.

Outline

- **The Insertion Sort Algorithm**
- **Insertion Sort - Arrays**
- **Time and Space Complexity of Insertion Sort**


Insertion Sort Algorithm 1

- The **lower** part of the collection is sorted and the **higher** part is unsorted.



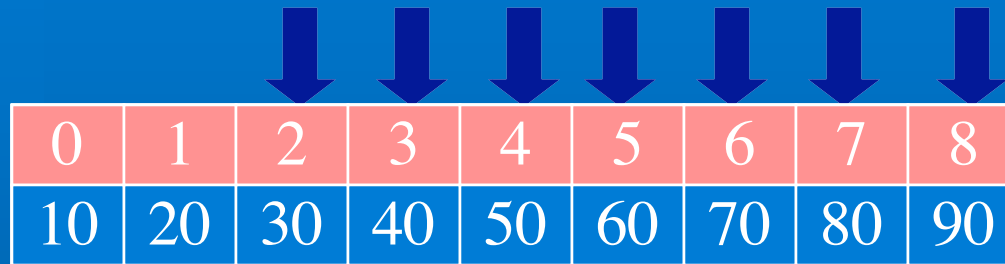
0	1	2	3	4	5	6	7	8
60	30	10	20	40	90	70	80	50

- Insert the first element of the unsorted part into the correct place in the sorted part.



0	1	2	3	4	5	6	7	8
30	60	10	20	40	90	70	80	50

Insertion Sort Algorithm 2



The diagram illustrates the Insertion Sort Algorithm 2. It features a horizontal array of nine cells. The top row of cells contains indices from 0 to 8, and the bottom row contains corresponding values: 10, 20, 30, 40, 50, 60, 70, 80, and 90. Above the array, a light blue horizontal bar spans from index 0 to index 5. Seven dark blue arrows point downwards from the top of the slide to the cells at indices 2, 3, 4, 5, 6, 7, and 8. The cells at indices 0 and 1 are not highlighted, while all other cells (indices 2-8) have a light red background.

0	1	2	3	4	5	6	7	8
10	20	30	40	50	60	70	80	90

Insertion Sort Code - Arrays

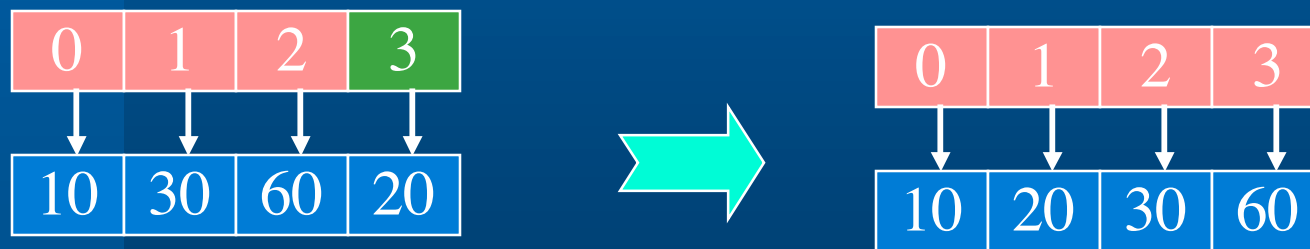
```
public static void insertionSort(Comparable
    anArray[], int size) {
    // pre: 0 <= size <= anArray.length
    // post: values in anArray are in ascending order

    int index; //index of start of unsorted part

    for (index = 1; index < size; index++) {
        moveElementAt(anArray, index);
    }
}
```

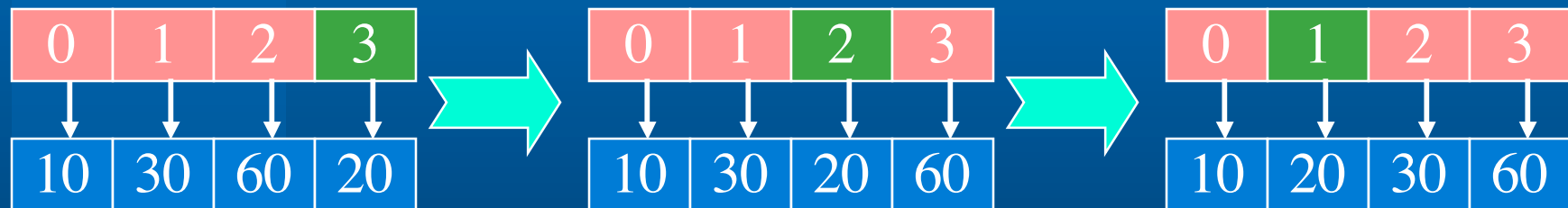
Moving Elements in Insertion Sort

- The Insertion Sort does not use an exchange operation.
- When an element is inserted into the ordered part of the collection, it is not just exchanged with another element.
- Several elements must be “moved”.



Multiple Element Exchanges

- The naïve approach is to just keep exchanging the new element with its left neighbor until it is in the right location.



- Every exchange costs four access operations.
- If we move the new element two spaces to the left, this costs $2 \times 4 = 8$ access operations.

Method - moveElementAt() - Arrays

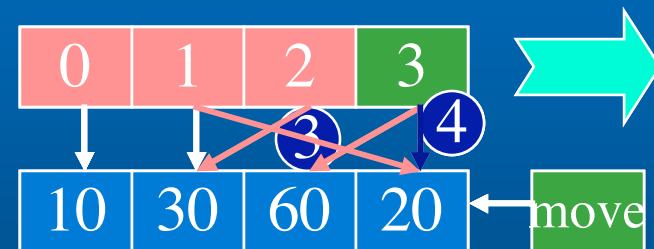
```
public static void moveElementAt(Comparable anArray[],
    int last) {
    // pre: 0 <= last < anArray.length and anArray in
    // ascending order from 0 to last-1
    // post: anArray in ascending order from 0 to last

    while ((last>0) &&
        (anArray[last].compareTo(anArray[last-1]) < 0)) {
        swap(anArray, last, last - 1);
        last--;
    }
}
```

Avoiding Multiple Exchanges

- We can insert the new element in the correct place with fewer accessing operations - only 6 accesses!

```
1. move = anArray[3];  
2. anArray[3] = anArray[2];  
3. anArray[2] = anArray[1];  
4. anArray[1] = move;
```



- In general if an element is moved (p) places it only takes $(2*p + 2)$ access operations, not $(4*p)$ access operations as required by (p) exchanges.

Recall Element Insertion in a Vector

- This operation is similar to inserting a new element in a Vector.
- Each existing element was “moved” to the right before inserting the new element in its correct location.

Recall Vector Insertion Code

```
public void insertElementAt(Object object, int index) {  
    //pre: 0 <= index <= size()  
    //post: inserts the given object at the given index,  
    // moving elements from index to size()-1 to the right  
  
    int i;  
  
    this.ensureCapacity(this.elementCount + 1);  
    for (i = this.elementCount; i > index; i--)  
        this.elementData[i] = this.elementData[i - 1];  
    this.elementData[index] = object;  
    this.elementCount++;  
}
```

Differences from Element Insertion

- **In Vector element insertion:**
 - We have a reference to the new element.
 - We know the index location for the new element.
- **In the Insertion sort:**
 - We don't have a reference to the new element, only an index in the array where the new element is currently located.
 - We don't know the index location for the new element. We need to find the index by comparing the new element with the elements in the collection from right to left.

Method - moveElementAt() - Arrays

```
public static void moveElementAt(Comparable anArray[],
    int last) {
    // pre: 0 <= last < anArray.length and anArray in
    // ascending order from 0 to last-1
    // post: anArray in ascending order from 0 to last

    Comparable move; //A reference to the element being moved

    move = anArray[last];
    while ((last>0) && (move.compareTo(anArray[last-1]) < 0)) {
        anArray[last] = anArray[last - 1];
        last--;
    }
    anArray[last] = move;
}
```

Counting Comparisons

- How many comparison operations are required for an insertion sort of an n -element collection?
- The sort method calls `moveElementAt()` in a loop for the indexes: $i = 1, 2, \dots, n - 1$.

```
for (index = 1; index < size; index++) {  
    this.moveElementAt(anArray, index);  
}
```

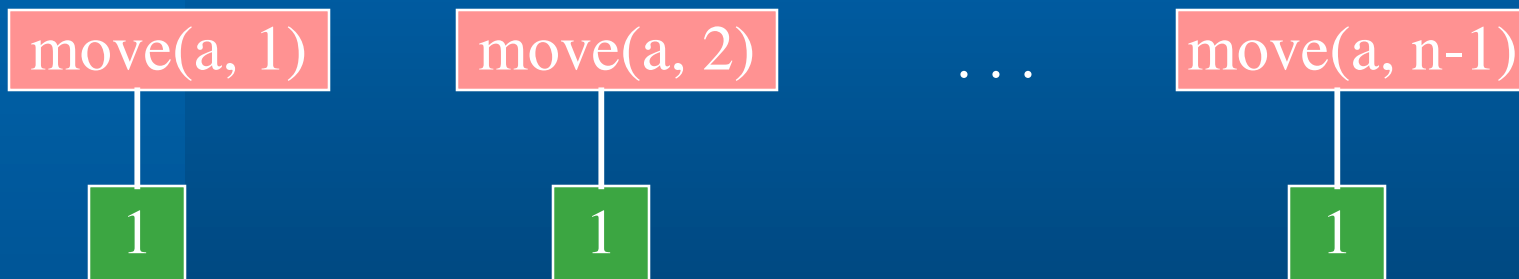
- Each time `moveElementAt()` is executed for some argument, `last`, it does a comparison in a loop for some of the indexes: `last, last-1, ... 1`.

```
while ((last > 0) && (anArray[last].compareTo(anArray[last-1]) < 0)) {  
    anArray[last] = anArray[last - 1];  
    last--;  
}
```

Comparisons - Best Case

- In the best case there is 1 comparison per call since the first comparison terminates the loop.

```
while ((last>0) && (anArray[last].compareTo(anArray[last-1])< 0)){  
    anArray[last] = anArray[last - 1];  
    last--;  
}
```



- The total number of comparisons is:
 $(n - 1) * 1 = n - 1 = O(n)$

Comparisons - Worst Case

- In the worst case there are "last" comparisons per call since the loop is not terminated until `last == 0`.

```
while ((last>0) && (anArray[last].compareTo(anArray[last-1])< 0)) {  
    anArray[last] = anArray[last - 1];  
    last--;  
}
```

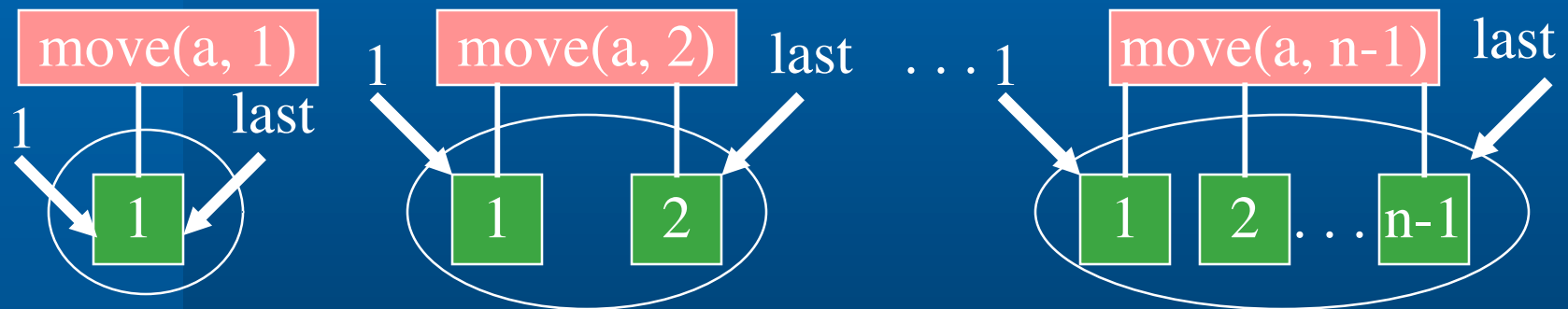


- The total number of comparisons is:
 $1 + 2 + \dots (n - 1) = [(n-1)*n] / 2 = O(n^2)$

Comparisons - Average Case 1

- In the average case it is equally probable that the number of comparisons is any number between 1 and "last" (inclusive) for each call.

```
while ((last>0) && (anArray[last].compareTo(anArray[last-1])< 0)) {
    anArray[last] = anArray[last - 1];
    last--;
}
```



- Note that the average for each call is:

$$(1 + 2 + \dots + \text{last}) / \text{last} = [\text{last} * (\text{last} + 1)] / [2 * \text{last}] = (\text{last} + 1) / 2$$

Comparisons - Average Case 2

- In the average case, the total number of comparisons is:

$$\begin{aligned}
 &(1+1)/2 + (2+1)/2 + \dots + ((n-1) + 1)/2 = \\
 &1/2 + 1/2 + 2/2 + 1/2 + \dots + (n-1)/2 + 1/2 = \\
 &[1 + 2 + \dots + (n-1)]*(1/2) + (n-1)*(1/2) = \\
 &[(n-1)*n/2]*(1/2) + (n-1)*(1/2) = \\
 &[(n-1)*n]*(1/4) + 2*(n-1)*(1/4) = \\
 &[(n-1)*n + 2*(n-1)]*(1/4) = \\
 &[(n-1)*(n + 2)]*(1/4) = O(n^2)
 \end{aligned}$$

Counting Moves

- How many move operations are required for an insertion sort of an n -element collection?
- The sort method calls `moveElementAt()` in a loop for the indexes: $k = 1, 2, \dots n - 1$.
- Every time the method is called, the element is moved one place for each successful comparison.

```
while ((last>0) && (anArray[last].compareTo(anArray[last-1]) < 0)){  
    anArray[last] = anArray[last - 1];  
    last--;  
}
```

- There is one move operation for each comparison so the best, worst and average number of moves is the same as the best, worst and average number of comparisons.

Counting Accesses

- Each comparison requires 2 accesses.
- Each move requires 2 accesses.
- Each time that `moveElementAt()` is called, there are two other accesses, one before the while loop and one after.
- Since `moveElementAt()` is called $n-1$ times, there are $2*(n-1) = O(n)$ of these extra accesses.
- Therefore, the "order" of the best, worst and average number of accesses is the same as the "order" of the best, worst and average number of comparisons.

Time Complexity of Insertion Sort

- Best case $O(n)$ accesses.
- Worst case $O(n^2)$ accesses.
- Average case $O(n^2)$ accesses.
- Note: this means that for nearly sorted collections, insertion sort is better than selection sort even though in average and worst cases, they are the same: $O(n^2)$.

Space Complexity of Insertion Sort

- Besides the collection itself, the only extra storage for this sort is the single temp reference used in the move element method.
- Therefore, the space complexity of Insertion Sort is $O(n)$.