# Algorithms and Data Structures (CSci 115)


California State University Fresno

College of Science and Mathematics

Department of Computer Science

H. Cecotti

# Learning Objectives

- What does it mean to sort an array?
  - Formal description
- Sorting algorithms:
  - Selection Sort
  - Insertion Sort
  - Bubble Sort
- Efficiency of Sorts

# Sorting / Ordering

- Very important to be able to sort data according to a **particular** criterion
- When data is ordered
  - → easier to locate some particular piece of data
- Once data is sorted, we can easily search in our data for a particular value
- Important to sort data as EFFICIENTLY as possible
- In most of the examples provided, we illustrate the PRINCIPLES of the technique using an array of integer values
- These basic principles are extensible to characters, string etc.
- Sorting and Searching go hand-in-hand but are very different

# Selection Sort

- Simple technique to both understand and implement
- We start with an UN-ORDERED array of integer values
- Technique operates via a series of PASSES
- In each pass,
  - we re-position a SINGLE value into its correct position with respect to the FINAL ORDERED ARRAY
- The main idea:
  - Swap to the smallest value and the value where the smallest value should go.

# Initial Array and Final Array

- 

**Initial UNSORTED Array**

| myArray | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| | 67 | 96 | 45 | 34 | 78 | 23 | 56 | 89 |

**Final SORTED Array**

| myArray | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| | 23 | 34 | 45 | 56 | 67 | 78 | 89 | 96 |

# Pass 1

- **In Pass 1**
  - We locate the SMALLEST ELEMENT in the array
  - We SWAP that value into its CORRECT FINAL POSITION in the array
- Note:
  - We need to use a Search algorithm to help us locate the smallest element
  - We can use a Linear Search to do this
- **Question**
  - Where will the smallest value be found in the final (sorted) array?
- **Answer**
  - Location 0
- Note that the repositioning is done using a SWAP

# Start Array and Finish Array

- 

**Initial UNSORTED Array**

| myArray | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| | 67 | 96 | 45 | 34 | 78 | 23 | 56 | 89 |

**Final SORTED Array**

| myArray | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| | 23 | 34 | 45 | 56 | 67 | 78 | 89 | 96 |

# Pass 1: Locate & Swap

**SWAP THESE 2 VALUES**

| myArray | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| | 67 | 96 | 45 | 34 | 78 | 23 | 56 | 89 |

0 `minPosition`

| myArray | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| | 23 | 96 | 45 | 34 | 78 | 67 | 56 | 89 |

# Situation AFTER Pass 1

- Element 0
  - Is now in its correct final position
  - It is ordered (sorted)
  - **It takes no further part in proceedings**
- Array has 2 parts:
  - ORDERED PART – 1 element i.e. Element 0
  - UNORDERED PART – 7 elements i.e. Elements 1 to 7

| myArray | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| | 23 | 96 | 45 | 34 | 78 | 67 | 56 | 89 |

# After Pass 1

- 

| myArray | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| | 23 | 96 | 45 | 34 | 78 | 67 | 56 | 89 |

Ordered Part of Array

Unordered Part of Array

# Pass 2: Repeat the Process

- In each Pass we repeat the process
- However, we restrict our work to the UNORDERED (UNSORTED) part of the array (only)
- We locate the smallest element in the unordered part of the array and SWAP it into it correct final position
- The technique used at each pass is identical

# Pass 2 – Locate & Swap

**SWAP THESE 2 VALUES**

| myArray | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| | 23 | 96 | 45 | 34 | 78 | 67 | 56 | 89 |

`1` `minPosition`

| myArray | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| | 23 | 34 | 45 | 96 | 78 | 67 | 56 | 89 |

# Situation AFTER Pass 2

- Elements 0 and 1
  - Are now ordered (sorted)
  - Take no further part in proceedings
- Array has 2 parts:
  - ORDERED PART – 2 elements i.e. 0 and 1
  - UNORDERED PART – 6 elements i.e. 2 thru 7

| myArray | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| | 23 | 34 | 45 | 96 | 78 | 67 | 56 | 89 |

# After Pass 2

| myArray | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| | 23 | 34 | 45 | 96 | 78 | 67 | 56 | 89 |

Ordered Part of Array

Unordered Part of Array

**Note that after each PASS, the ORDERED PART of the array grows while the UNORDERED PART of the array shrinks**

# Pass 3: Repeat the Process

- Repeat the process using the UNORDERED/UNSORTED part of the array (only)
- Locate the smallest element in the unordered part of the array and SWAP it into it correct final position

# Pass 3 - Interesting

SWAP THESE 2 VALUES

| myArray | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| | 23 | 34 | 45 | 96 | 78 | 67 | 56 | 89 |

2 `minPosition`

| myArray | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| | 23 | 34 | 45 | 96 | 78 | 67 | 56 | 89 |

# Some Specific Thoughts

- In Pass 3
  - the array element just happens to already be in its correct position
- This is reasonably common – there is a good chance that this will happen **at least once** during a particular Selection Sort
- However, we cannot guarantee this so we still must go through the entire process
- On completion of the pass, the ordered part of the array will increase in size (by 1 element) and the unordered part will decrease (by 1 element)

# Generally ….

- In general, in:
  - 1 PASS we have 1 element correctly ordered
  - 2 PASSES we have 2 elements correctly ordered
  - 3 PASSES we have 3 elements correctly ordered

    …..
  - N PASSES we have N elements correctly ordered
- However in practice, it clearly only takes:
  - 7 PASSES to correctly order an Array of 8 elements
  - 11 PASSES to correctly order an Array of 12 elements
  - N-1 PASSES to correctly order an Array of N elements

# Insertion Sort

- Operates as series of Passes
- After each Pass we have move closer to a solution
- Note that we do NOT CORRECTLY POSITION values in a pass
- We do, however, CORRECTLY ORDER elements in a portion of the array
- The main idea
  - Swap the values if one is smallest than the current one.

# What are we doing?

- What we actually do is ensure that after:
- **Pass 1**
  - First 2 elements are in their CORRECT ORDER
- **Pass 2**
  - First 3 elements are in their CORRECT ORDER
- **Pass 3**
  - First 4 elements are in their CORRECT ORDER

# Initial Array and Final Array

- **Initial UNSORTED Array**

| myArray | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| | 79 | 61 | 50 | 69 | 18 | 57 | 28 |

**Final SORTED Array**

| myArray | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| | 18 | 28 | 50 | 57 | 61 | 69 | 79 |

# Pass 1: Correctly order elements 1 & 2

- **Initial UNSORTED Array**

| myArray | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| | 79 | 61 | 50 | 69 | 18 | 57 | 28 |

The element under consideration is in location 1 (61)
We are going to CORRECTLY ORDER elements 0 and 1
We do this by:

(i) SAVING the value at location 1
(ii) Advancing (or promoting) any values that are LARGER than this saved value
(iii) Finally re-insert the saved value at its appropriate position

# Best to think as follows:

In detail:

- When we SAVE the value at location 1, we simply place the value in location 1 in a temporary **`int`** variable (called **`saved`**)

```
int saved = myArray[1];
```

# Best to think as follows:

- When we say ADVANCE LARGER VALUES - we need code to perform this, such as:

```
myArray[1] = myArray[0];
```

| myArray | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| | 79 | 79 | 50 | 69 | 18 | 57 | 28 |

# Best to think as follows:

- Finally INSERT the saved value
- Use code similar to:

```
myArray[0] = saved;
```

| myArray | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| | 61 | 79 | 50 | 69 | 18 | 57 | 28 |

# After Pass 1:
# First 2 elements are in order

- 

| myArray | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| | 61 | 79 | 50 | 69 | 18 | 57 | 28 |

# Pass 2: Correctly order elements 0, 1 & 2

▪

**Initial UNSORTED Array**

| myArray | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| | 61 | 79 | 50 | 69 | 18 | 57 | 28 |

- Element under consideration is in location 2
- We are now going to correctly order elements 0, 1 and 2

# Pass 2: Correctly order elements 0, 1 & 2

- Insert Element at location 2 (i.e. 50) into the early part of the array so that elements 0, 1 and 2 are in order

| myArray | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| | 61 | 79 | **50** | 69 | 18 | 57 | 28 |

# Pass 2: Correctly order elements 0, 1 & 2

- Do this in the same manner as previous:

  (i) SAVE the value currently at location 2

  (i) Then advance (promote) any larger values

  (i) Then re-insert the saved value at the appropriate position

| myArray | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| | 61 | 79 | **50** | 69 | 18 | 57 | 28 |

# Pass 2 – In this case ….

| myArray | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| | **50** | **61** | **79** | 69 | 18 | 57 | 28 |

- In this particular case, we promote 2 values
- Finally, 50 is inserted at location 0
- Note that 3 elements are now correctly ordered

# Pass 3

| myArray | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| | 50 | 61 | 79 | **69** | 18 | 57 | 28 |

- In Pass 3 we consider the element currently at index position 3 and save it
- We then “advance” values larger than the saved value
- Then we re-insert the saved value

## AFTER 3 PASSES

| myArray | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| | 50 | 61 | 69 | 79 | 18 | 57 | 28 |

ESSENTIALLY - after 3 passes, 4 elements will be in their correct order

## AFTER 4 PASSES

| myArray | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| | 18 | 50 | 61 | 69 | 79 | 57 | 28 |

## AFTER 5 PASSES

| myArray | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| | 18 | 50 | 57 | 61 | 69 | 79 | 28 |

## AFTER 6 PASSES

| myArray | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| | 18 | 28 | 50 | 57 | 61 | 69 | 79 |

# Bubble Sort

- A Sorting Technique that also works on a series of passes
- In each PASS:
  - We compare adjacent overlapping pairs of elements to make sure they are in the CORRECT ORDER with respect to the final array
- NOTE – we are only interested in whether they are in the correct relative order and NOT necessarily their correct FINAL POSITION

# Adjacent Overlapping Pairs?

- Adjacent *overlapping* pairs of elements are as follows:

    Elements 0 and 1

    Elements 1 and 2

    Elements 2 and 3

    Elements 3 and 4

    ……

    Elements (N-2) and (N-1)

    Elements (N-1) and N

# Start Array and Finish Array

- **Initial UNSORTED Array**

| myArray | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| | 67 | 96 | 45 | 34 | 78 | 23 | 56 | 89 |

**Final SORTED Array**

| myArray | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| | 23 | 34 | 45 | 56 | 67 | 78 | 89 | 96 |

# Pass 1 – Swap Pairs ‘Out of Order’

- **COMPARE FIRST ADJACENT PAIR OF ELEMENTS**

| myArray | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| | 67 | 96 | 45 | 34 | 78 | 23 | 56 | 89 |

IN ORDER
SO
NO SWAP

# Pass 1 – Swap Pairs ‘Out of Order’

- COMPARE NEXT ADJACENT PAIR OF ELEMENTS

| myArray | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| | 67 | 96 | 45 | 34 | 78 | 23 | 56 | 89 |

**Note that 96 and 45 will be swapped**

OUT OF ORDER SO SWAP

# Pass 1 – Swap Pairs ‘Out of Order’

- 

**COMPARE NEXT ADJACENT PAIR OF ELEMENTS**

| myArray | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| | 67 | 45 | 96 | 34 | 78 | 23 | 56 | 89 |

**Note that 96 and 34 will be swapped**

OUT OF ORDER SO SWAP

# Pass 1 – Swap Pairs ‘Out of Order’

**COMPARE NEXT ADJACENT PAIR OF ELEMENTS**

| myArray | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| | 67 | 45 | 34 | 96 | 78 | 23 | 56 | 89 |

OUT OF ORDER SO SWAP

**Note that 96 and 78 will be swapped**

# Pass 1 – Swap Pairs 'Out of Order'

- 

**COMPARE NEXT ADJACENT PAIR OF ELEMENTS**

| myArray | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| | 67 | 45 | 34 | 78 | 96 | 23 | 56 | 89 |

OUT OF ORDER SO SWAP

**Note that 96 and 23 will be swapped**

# Pass 1 – Swap Pairs ‘Out of Order’

▪

**COMPARE NEXT ADJACENT PAIR OF ELEMENTS**

| myArray | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| | 67 | 45 | 34 | 78 | 23 | 96 | 56 | 89 |

OUT OF ORDER SO SWAP

**Note that 96 and 56 will be swapped**

# Pass 1 – Swap Pairs ‘Out of Order’

■

COMPARE NEXT ADJACENT PAIR OF ELEMENTS

| myArray | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| | 67 | 45 | 34 | 78 | 23 | 56 | 96 | 89 |

OUT OF ORDER SO SWAP

**Note that 96 and 89 will be swapped**

# PASS 1 – COMPLETED!

| myArray | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| | 67 | 45 | 34 | 78 | 23 | 56 | 89 | 96 |

**Unordered Part of Array**

**Ordered Part of Array**

The ORDERED PART of Array consists of just 1 element which takes no further part in proceeding

# End of Pass 1

| myArray | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| | 67 | 45 | 34 | 78 | 23 | 56 | 89 | 96 |

At the end of PASS 1, 1 element, the largest value has worked its way to the end of the array

We say it has **BUBBLED** its way to its CORRECT FINAL POSITION in the array

# Continue in this manner

- We continue in this manner comparing adjacent pairs of elements
- We swap pairs that are ‘out of order’
- We do NOT move pairs that are ‘in order’

# Repeat the Process

- We continue in this manner for a series of passes
- In each pass we correctly reposition 1 element
- Hence in PASS 2 our array will look as follows:

**ARRAY AT THE END OF PASS 2**

| myArray | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| | 45 | 34 | 67 | 23 | 56 | 78 | 89 | 96 |

Unordered Part of Array

Ordered Part of Array

# Continue to completion

- In each pass we correctly position 1 element
- As before when we have 8 elements we require a maximum of 7 passes
- If 7 of the 8 elements are in their correct final position then all 8 must be in their correct final position

# Efficiency - Detect a Quick Finish?

- With the Bubble Sort Technique we can detect an ‘early sort’
- How??
    - **If, within a given pass, NO SWAPS are required**
- Think about it…:
    - No Swaps in a particular pass implies that each adjacent pair of elements is in their correct (relative) position

# Efficiency – Selection Sort

- For 10 items

  PASS 1 – inner loop makes (10 – 1) = 9 comparisons
  PASS 2 – inner loop makes (10 – 2) = 8 comparisons
  PASS 3 – inner loop makes (10 – 3) = 7 comparisons
  ….
  Total comparisons for 10 items

  $$9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1 = 45$$

- In general, for N items the number of comparisons is always

  $$(N-1) + (N-2) + (N-3) + \ldots + 1 = N*(N-1)/2$$

  - For 10 items we require a maximum of 9 swaps
  - For 100 items, 4,950 comparisons are required, but a maximum of 99 swaps
  - For large values of N, the number of comparisons will dominate

# Efficiency – Insertion Sort

- For 10 items

  PASS 1 – maximum of 1 comparison
  PASS 2 – maximum of 2 comparisons
  PASS 3 – maximum of 3 comparisons
  ….
  Total comparisons for 10 items

  $$1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 = 45$$

- In general, for N items the number of comparisons is

  $$1 + 2 + 3 + 4 + 5 + \ldots + (N-1) = N*(N-1)/2$$

- At each PASS only half (on average) of the maximum number of items are compared, the actual number of comparisons is

  $$N * (N-1)/4$$

  - The number of copies is approximately the same as the number of comparisons
  - If data is sorted there are only (N-1) comparisons and no copies

# Efficiency – Bubble Sort

- For 10 items

  PASS 1 – maximum of 1 comparison
  PASS 2 – maximum of 2 comparisons
  PASS 3 – maximum of 3 comparisons
  ….
  Total comparisons for 10 items

  $$1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 = 45$$

- In general, for N items the number of comparisons is

  $$1 + 2 + 3 + 4 + 5 + \ldots + (N-1) = N*(N-1)/2$$

  - you can prove it through induction with a recursive function
  - The number of swaps is approximately half the number of comparisons
  - If data is sorted there are only (N-1) comparisons and no swaps

# Conclusion

- Presentation of sorting algorithms
  - $O(n^2)$
  - → It s possible to find better ☺

- Questions?