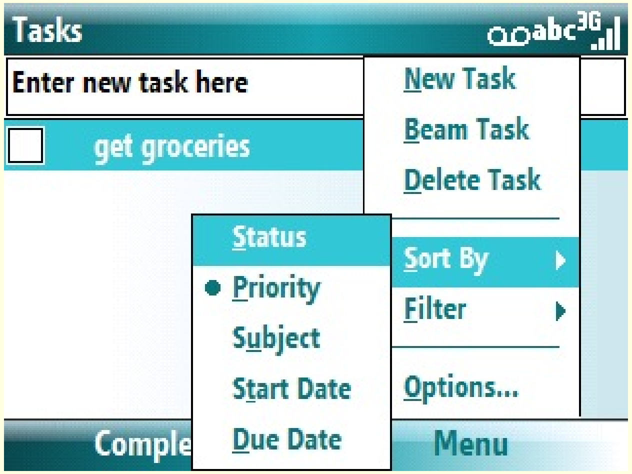
Sorting: $O(n^2)$ Algorithms

Computer Science Department
University of Central Florida

COP 3502 – Computer Science I

Sorting: O(n^2) Algorithms

- Sorting Algorithms:
  - Fundamental problem in Computer Science
  - Sorting is done to make searching easier
  - Most programs do this:
    - Excel, Access, and others.



Sorting: $O(n^2)$ Algorithms

- Sorting Algorithms:
  - We will study several sorting algorithms in this class
    - Some are clearly much faster than others
  - For today, we will go over the “simple sorts”
  - These “simple sorts” all run in $O(n^2)$ time
    - Selection Sort
    - Insertion Sort
    - Bubble Sort
    - We will assume that the input to the algorithm is an array of values (sorted or not)

Sorting: $O(n^2)$ Algorithms

- **<u>Selection Sort:</u>**
  - Given: an array of n unsorted items
  - The algorithm to sort n numbers is as follows:
    1) Find the minimum value in the list of n elements
       - Search from index 0 to index n-1
    2) Swap that minimum value with the value in the first position
       - At index 0
    3) Repeat steps 1 and 2 for the remainder of the list
       - Example:
       - We now start at the 2nd position (index 1).
       - Find minimum value from index 1 to index n-1
       - Swap that minimum value with the value at index 1

Sorting: $O(n^2)$ Algorithms

- **Selection Sort:**
  - The algorithm to sort n numbers is as follows:
    - There is a FOR loop that iterates from i = 0 to i = n-1
    - FOR the i^{th} element (as i ranges from 0 to n-1)
    1) Determine the smallest element in the rest of the array
       - To the right of the i^{th} element
    2) Swap the current i^{th} element with the element identified in part (1) above (the smallest element)
    - Essentially:
      - The algorithm first picks the smallest element and swaps it into the first location.
      - Then it picks the next smallest element and swaps it into the next location, etc.

Sorting: O(n^2) Algorithms

- **Selection Sort:**
  - Example:
    - Here is an array of 5 integers

20	8	5	10	7
0	1	2	3	4

i = 0

- Remember, we have a for loop

```
FOR i = 0 to n – 1 {
    Find the minimum value in the range from i to n-1
    SWAP this minimum value with the value at index i
}
```

- NOTE: i represents the index into the array

Sorting: $O(n^2)$ Algorithms

- **Selection Sort:**
  - Example:
    - Here is an array of 5 integers

20	8	5	10	7
0	1	2	3	4

i = 0

    - 5 (at index 2) is the smallest element
      - from the range i = 0 to 4
    - So SWAP the value at index 2 with the value at index 0
      - SWAP the 5 and the 20

5	8	20	10	7
0	1	2	3	4

Sorting: O(n^2) Algorithms

- **Selection Sort:**
  - Example:
    - Here is an array of 5 integers

5	8	20	10	7
0	1	2	3	4

i = 1

    - 7 (at index 4) is the smallest element
      - from the range i = 1 to 4
    - So SWAP the value at index 4 with the value at index 1
      - SWAP the 7 and the 8

5	7	20	10	8
0	1	2	3	4

Sorting: $O(n^2)$ Algorithms

- **Selection Sort:**
  - Example:
    - Here is an array of 5 integers

0	1	2	3	4
5	7	20	10	8

i = 2

    - 8 (at index 4) is the smallest element
      - from the range i = 2 to 4
    - So SWAP the value at index 4 with the value at index 2
      - SWAP the 8 and the 20

0	1	2	3	4
5	7	8	10	20

Sorting: O(n^2) Algorithms

- **<u>Selection Sort:</u>**
  - Example:
    - Here is an array of 5 integers

5	7	8	10	20
0	1	2	3	4

i = 3

    - 10 (at index 3) is the smallest element
      - from the range i = 3 to 4
    - So SWAP the value at index 3 with the value at index 3
      - SWAP the 10 and the 10 (so no swap really happened here)

5	7	8	10	20
0	1	2	3	4

Sorting: $O(n^2)$ Algorithms

- **Selection Sort:**
  - Example:
    - Here is an array of 5 integers

5	7	8	10	20
0	1	2	3	4

i = 4

    - 20 (at index 4) is the smallest element
      - from the range i = 4 to 4
    - So SWAP the value at index 4 with the value at index 4
      - SWAP the 20 and the 20 (so no swap really happened here)

5	7	8	10	20
0	1	2	3	4

Sorting: $O(n^2)$ Algorithms

- **Selection Sort:**
  - Example:
    - Here is an array of 5 integers

5	7	8	10	20
0	1	2	3	4

i = 4

    - The array is now in sorted order
    - We see that the last iteration was not even necessary
      - In code our for loop could look like this:

```
for (i = 0; i < n-1; i++)
```

      - So it won't even iterate on the n-1 step

Sorting: O(n^2) Algorithms

- **<u>Selection Sort:</u>**
  - Analysis of Running Time:
    - During the first iteration
      - We "go through" all n items searching for the minimum
      - This is essentially n simple steps
    - During the second iteration, i starts at index 1
      - We "go through" n – 1 items searching for the minimum
      - We do not need to account for the item at index 0
      - Cuz it is already in the correct position!
    - During the third iteration,
      - We "go through" n – 2 items searching for the minimum
      - We do not need to account for the items at index 0 and 1
      - Cuz they are already in correct position

Sorting: $O(n^2)$ Algorithms

- **<u>Selection Sort:</u>**
  - Analysis of Running Time:
    - 4th iteration:
      - We will "go through" n – 3 steps
    - 5th iteration
      - We will "go through" n – 4 steps
    - …
    - Final iteration
      - There will simply be one step
    - We can add up the TOTAL number of simple steps
    - TOTAL = n + (n-1) + (n-2) + (n-3) + … + 3 + 2 + 1
    - Is this n^2 steps? Perhaps logn steps? Perhaps n steps?

Sorting: $O(n^2)$ Algorithms

- **<u>Selection Sort:</u>**
  - Analysis of Running Time:
    - TOTAL = n + (n-1) + (n-2) + (n-3) + … + 3 + 2 + 1
    - We does this add up to?
      - We need to know this in order to give the Big-O
    - There is a neat trick!
    - Write the equation shown above
    - And then immediately underneath,
      - Write the equation again, but REVERSE the order of the terms
    - Then add the two equations together
      - See what happens
    - Finally, solve for TOTAL

Sorting: $O(n^2)$ Algorithms

- **Selection Sort:**
  - Analysis of Running Time:

$$
\begin{array}{rl}
 & TOTAL = n + (n-1) + (n-2) + (n-3) + \ldots + 3 + 2 + 1 \\
+ & TOTAL = 1 + 2 + 3 + 4 + \ldots + (n-2) + (n-1) + n \\
\hline
 & 2*TOTAL = (n+1) + (n+1) + (n+1) + \ldots + (n+1) + (n+1)
\end{array}
$$

    - How many terms of (n+1) do we have?
      - We have n of them!
    - So that is n*(n+1)
    - 2*TOTAL = n(n+1)
    - TOTAL = n(n+1)/2
    - So we see that Selection sort runs in $O(n^2)$ time.

Sorting: $O(n^2)$ Algorithms

- **<u>Insertion Sort:</u>**
  - This is the sort that most humans apply when sorting documents
  - Example: Playing Cards
    - Players usually keep cards in sorted order
    - When you pick up a new card
      - You make room for the new card and put into its proper place

Sorting: O(n^2) Algorithms

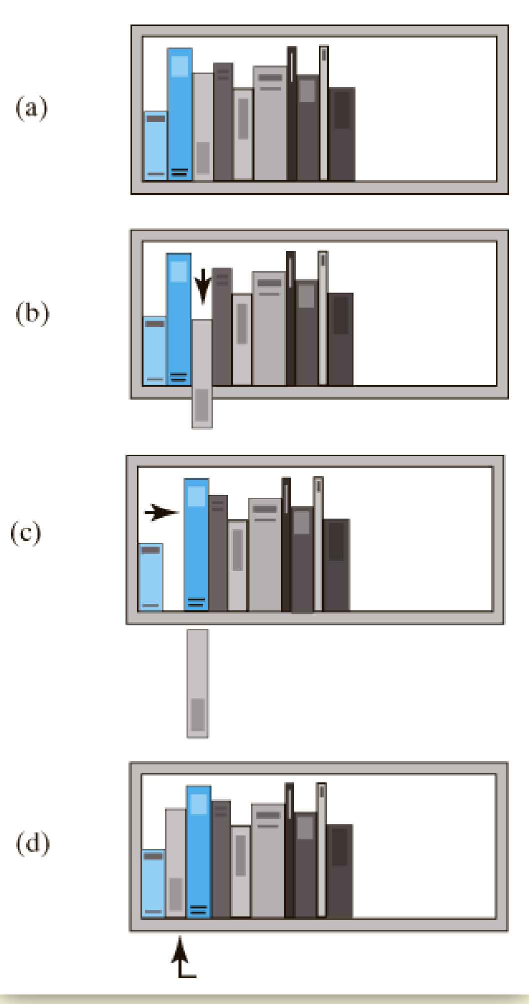
- **<u>Insertion Sort:</u>**
  - The card example demonstrates the basic idea of Insertion Sort
    - But the "idea" isn't exactly the same as sorting an array of items
  - When sorting an array of items, we are ALREADY holding all of the items
  - So how are we "inserting" an item when it is already in the list.
  - We remove the items, one at a time, and then reinsert them into their proper positions

Sorting: $O(n^2)$ Algorithms

- **Insertion Sort:**
  - Bookshelf example:
  - If first two books are out of order:
    - Remove second book
    - Slide first book to right
    - Insert removed book into first slot
  - Next, look at third book, if it is out of order:
    - Remove that book
    - Slide 2nd book to right
    - Insert removed book into 2nd slot
  - Recheck first two books again
    - Etc.

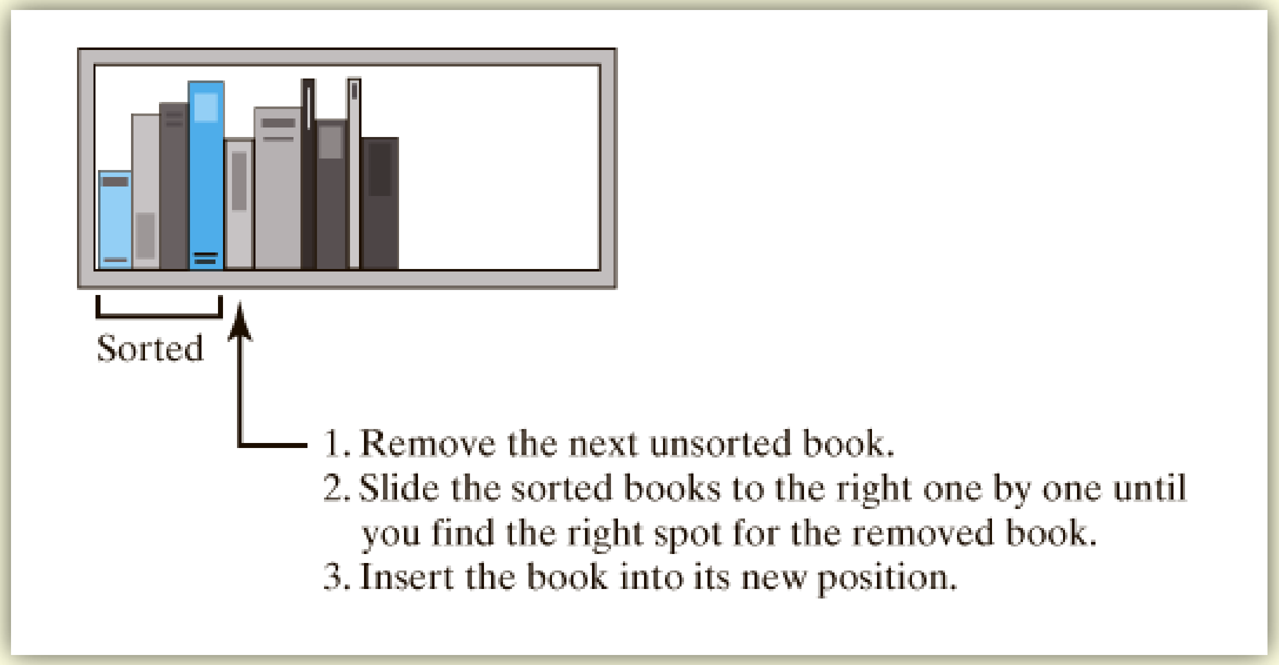
Sorting: $O(n^2)$ Algorithms

- **Insertion Sort:**
  - Bookshelf example:
    - This picture shows the "insertion" of the third book
      - The 3rd book is removed
      - It is compared with the 2nd book
      - The 2nd book is larger
      - So we slide the 2nd book into the 3rd spot
      - We then compare our original 3rd book with the 1st book
      - They are in order
      - So we simply insert the original 3rd book in the 2nd spot



Sorting: $O(n^2)$ Algorithms

- **<u>Insertion Sort:</u>**
  - Bookshelf example:
    - In general:



Sorting: $O(n^2)$ Algorithms

- **<u>Insertion Sort:</u>**
  - Given: an array of n unsorted items
  - The algorithm to sort n numbers is as follows:
    - Starting with the 2nd element,
    - Take each element, one by one, and
    - "Insert" it into a sorted list
    - How do we insert it?
      - continually SWAP it with the previous element until it has found its correct spot in the already sorted list
        - When we say already sorted list, we are referring to the elements to the left of our current element
        - Those elements are already in sorted order

Sorting: $O(n^2)$ Algorithms

- **Insertion Sort:**
  - The algorithm to sort n numbers is as follows:
    - For the ith element
      - as i ranges from 1 to n-1 (week skip i = 0, the 1st element)
    - As long as the current element is smaller than the element before it
      - SWAP the two elements
    - Stop when the current element is bigger than the one before it OR there is no element before it
      - Meaning it has reached the front

    - An example should clarify…

Sorting: O(n^2) Algorithms

- **<u>Insertion Sort:</u>**
  - Example:
    - Here is an array of 5 integers

3	7	2	1	5
0	1	2	3	4

i = 1

    - Remember, we have a for loop

FOR i = 1 to n – 1 {

WHILE the current element (at index i) is smaller than the element before it

SWAP the two elements

}

    - NOTE: i represents the index into the array

Sorting: O(n^2) Algorithms

- **<u>Insertion Sort:</u>**
  - Example:
    - Here is an array of 5 integers

3	7	2	1	5
0	1	2	3	4

i = 1

    - 7 is the value at index 1
      - Compare 7 to the value at index 0 (which is 3)
    - 7 is greater than 3
      - So there is nothing to swap. Simply re-insert 7 at its place.

3	7	2	1	5
0	1	2	3	4

Sorting: O(n^2) Algorithms

- **<u>Insertion Sort:</u>**
  - Example:
    - Here is an array of 5 integers

3	7	2	1	5
0	1	2	3	4

i = 2

    - 2 is the value at index 2
      - Compare 2 to the value at index 1 (which is 7)
    - 2 is smaller than 7
      - So we SWAP
    - BUT we are NOT done!

Sorting: $O(n^2)$ Algorithms

- **Insertion Sort:**
  - Example:
    - Here is an array of 5 integers

3	7	2	1	5
0	1	2	3	4

i = 2

    - We must compare the 2 to the value at index 0
      - which is 3
    - 2 is smaller than 3
      - So we SWAP

2	3	7	1	5
0	1	2	3	4

Sorting: $O(n^2)$ Algorithms

- **<u>Insertion Sort:</u>**
  - Example:
    - Here is an array of 5 integers

2	3	7	1	5
0	1	2	3	4

i = 3

    - 1 is the value at index 3
      - Compare 1 to the value at index 2 (which is 7)
    - 1 is smaller than 7
      - So we SWAP
    - 1 is smaller than 3
      - So we SWAP

Continue comparing 1 to the element before it

Sorting: $O(n^2)$ Algorithms

- **<u>Insertion Sort:</u>**
  - Example:
    - Here is an array of 5 integers

2	3	7	1	5
0	1	2	3	4

i = 3

    - 1 is smaller than the value at index 0 (which is 2)
      - So we SWAP
    - There is no element "before" 1 at this point
      - So we simply insert

1	2	3	7	5
0	1	2	3	4

Sorting: $O(n^2)$ Algorithms

- **<u>Insertion Sort:</u>**
  - Example:
    - Here is an array of 5 integers

1	2	3	7	5
0	1	2	3	4

i = 4

    - 5 is the value at index 4
      - Compare 5 to the value at index 3 (which is 7)
    - 5 is smaller than 7
      - So we SWAP
    - Again, we are not done

Sorting: $O(n^2)$ Algorithms

- **<u>Insertion Sort:</u>**
  - Example:
    - Here is an array of 5 integers

1	2	3	7	5
0	1	2	3	4

i = 4

    - We must now compare 5 to the next element before it
    - So compare 5 to 3
    - 5 is greater than 3, so we can stop and insert 5

1	2	3	5	7
0	1	2	3	4

Sorting: O(n^2) Algorithms

- **<u>Insertion Sort:</u>**
  - Analysis of Running Time:
    - The number of steps varies based on the input
    - If the list is already in sorted order (best case)
      - During each iteration, the ith element is only compared with one previous element
      - This results in a linear run-time, or O(n)
    - If the list is sorted in reverse order (worst case)
      - During each iteration, the ith element will have to go all the way over to the left
        - During each iteration, the entire, sorted subsection of the array will be shifted over to allow the ith element to go into the front
        - This results in a quadratic run-time, or O(n^2)
    - We care about worst case; Insertion Sort runs in O(n^2).

Brief Interlude: FAIL Picture

Sorting: O(n^2) Algorithms

- **<u>Bubble Sort:</u>**
  - Basic idea:
    - You always compare consecutive elements
      - Going left to right
    - Whenever two elements are out of place,
      - SWAP them
    - At the end of a single iteration,
      - the maximum element will be in the last spot
    - Now you simply repeat this n times
      - where n is the number of elements being sorted
    - One each pass, one more maximal element will be put into place

Sorting: $O(n^2)$ Algorithms

- **<u>Bubble Sort:</u>**
  - Example:
    - Here is an array of 8 integers: 6, 2, 5, 7, 3, 8, 4, 1
    - On a single pass of the algorithm, here is the state of the array:

<u>2, 6</u>, 5, 7, 3, 8, 4, 1
2, <u>5, 6</u>, 7, 3, 8, 4, 1
2, 5, <u>6, 7</u>, 3, 8, 4, 1
2, 5, 6, <u>3, 7</u>, 8, 4, 1
2, 5, 6, 3, <u>7, 8</u>, 4, 1
2, 5, 6, 3, 7, <u>4, 8</u>, 1
2, 5, 6, 3, 7, 4, <u>1, 8</u> (8 is now in place!)

The “swapped” elements are underlined.

Of course, a swap only occurs as needed.

Sorting: $O(n^2)$ Algorithms

- **Bubble Sort:**
  - Truth about Bubble Sort:
  - NOBODY uses Bubble Sort
  - NOBODY.
  - EVER.
  - ‘cept this guy:

  - cuz Bubble sort is extremely inefficient

Sorting: O(n^2) Algorithms

- **<u>Sorts that only swap adjacent elements</u>**
  - Selection, Insertion, and Bubble sort are examples of sorts where we swap adjacent elements
  - LIMITATION of these types of sorts:
    - They can only run so fast.
  - We can see this once we define an inversion:
    - Inversion: a pair of numbers in a list that is out of order
    - Given this list: 3, 1, 8, 4, 5
    - The inversions are the following pairs of numbers:
      - (3,1), (8, 4), and then (8, 5)

Sorting: $O(n^2)$ Algorithms

- **<u>Sorts that only swap adjacent elements</u>**
  - LIMITATION of these types of sorts:
    - They can only run so fast.
  - We can see this once we define an inversion:
    - When we <u>swap adjacent elements</u> in an array
      - We can remove AT MOST one inversion from the array
    - Now, if it were possible to <u>swap non-adjacent elements</u>,
      - We could remove multiple inversions at the same time
      - Consider the following list: 8, 2, 3, 4, 5, 6, 7, 1
        - Only 8 and 1 are out of order
        - Swapping these two values would remove every inversion
        - It would normally require 13 inversions to get the list sorted if we were limited to swapping only adjacent elements

Sorting: O(n^2) Algorithms

- **<u>Sorts that only swap adjacent elements</u>**
  - Run-time Analysis:
    - Any sorting algorithm that swaps adjacent elements is constrained by the total number of inversions in that array
    - Consider the average case:
      - How many pairs of numbers are there in a list of n numbers?

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

You learn this in Discrete and certain Math courses. For now, just trust me on this.

      - Of these pairs, on average, HALF of them will be inverted.

$$\frac{n(n-1)}{4}$$

We simply divided the previous amount by 2, thus leaving HALF of the pairs left.

Sorting: $O(n^2)$ Algorithms

- **<u>Sorts that only swap adjacent elements</u>**
  - Run-time Analysis:
    - So, on average, an unsorted array will have

      $\frac{n(n-1)}{4}$ inversions

    - **<u>Therefore, any sorting algorithm that only swaps adjacent elements will have an $O(n^2)$ run-time.</u>**

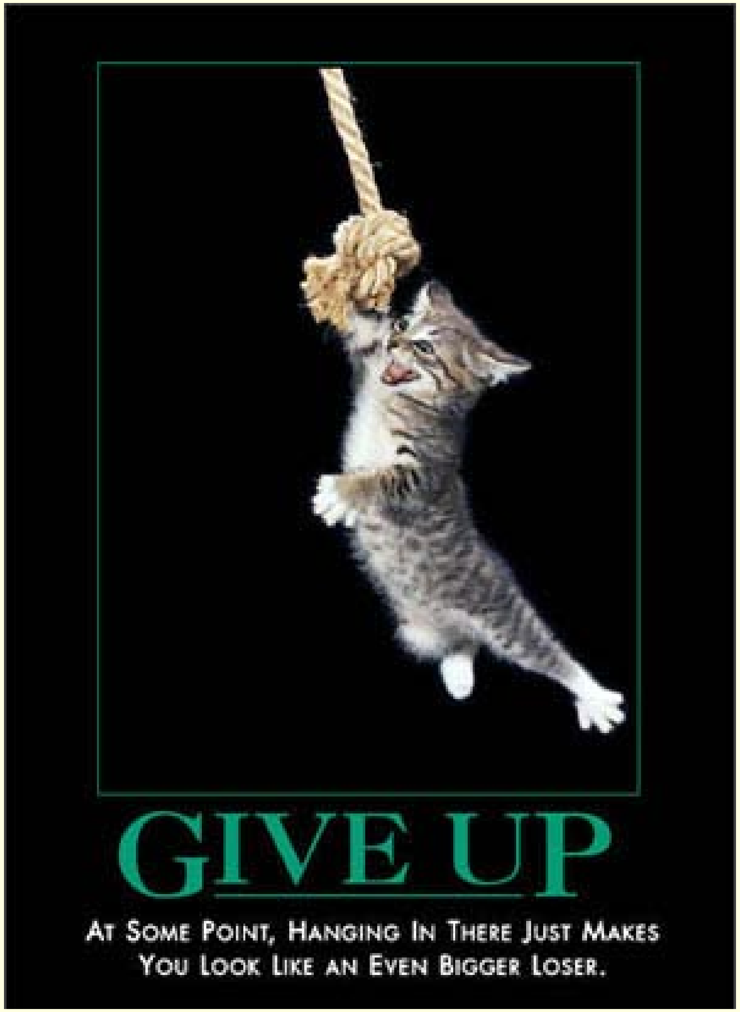
Sorting: O(n^2) Algorithms

WASN'T

THAT

AMAZING!

Daily Demotivator



Sorting: $O(n^2)$ Algorithms

Computer Science Department
University of Central Florida

COP 3502 – Computer Science I