Algorithms

Arthur Hoskey, Ph.D.
Farmingdale State College
Computer Systems Department

Today's Lecture

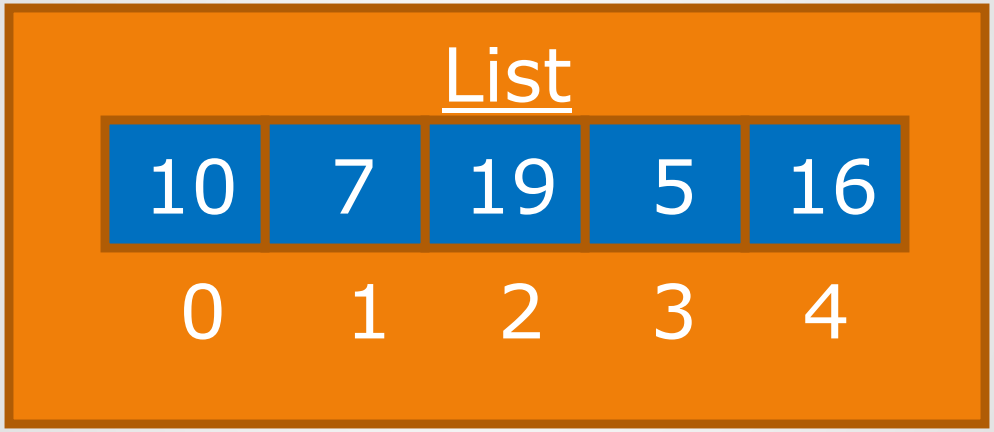
- Bubble sort

Bubble Sort

- Elementary sorting algorithm.
- The basic algorithm is to test adjacent elements and swap them if they are out of order.
- Bubble sort is slow in comparison to most other sorting algorithms.

Bubble Sort Overview

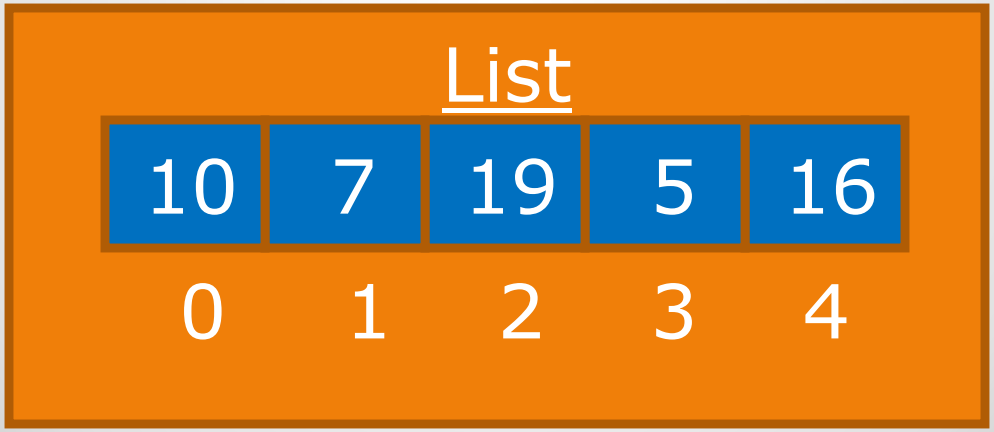
- Sort the following list using bubble sort.



Bubble Sort

Bubble Sort

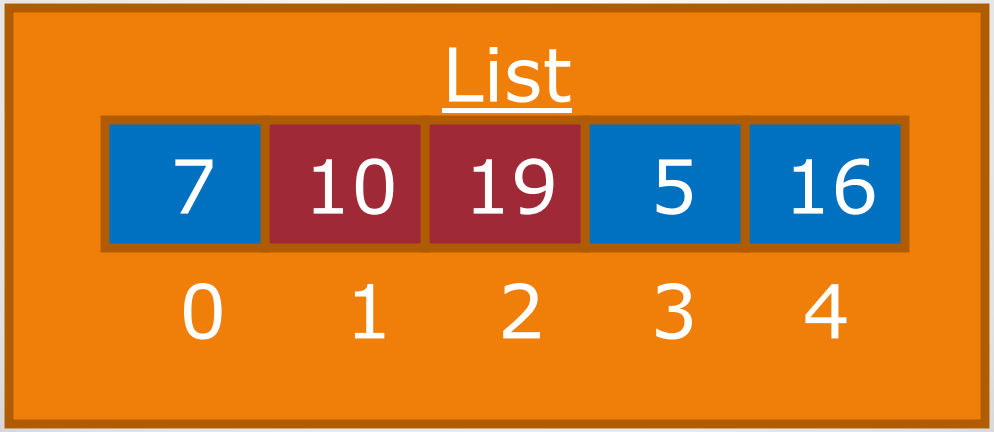
- Bubble sort.
- Go through the array from beginning to end.
- Check adjacent pairs. If the elements in a pair are out of order swap them.
- "Large" elements will bubble to the end of the array on each pass through the array.



Bubble Sort

- Pass 1
- Compare elements at index 0 and 1.
- These elements are out of order so swap.

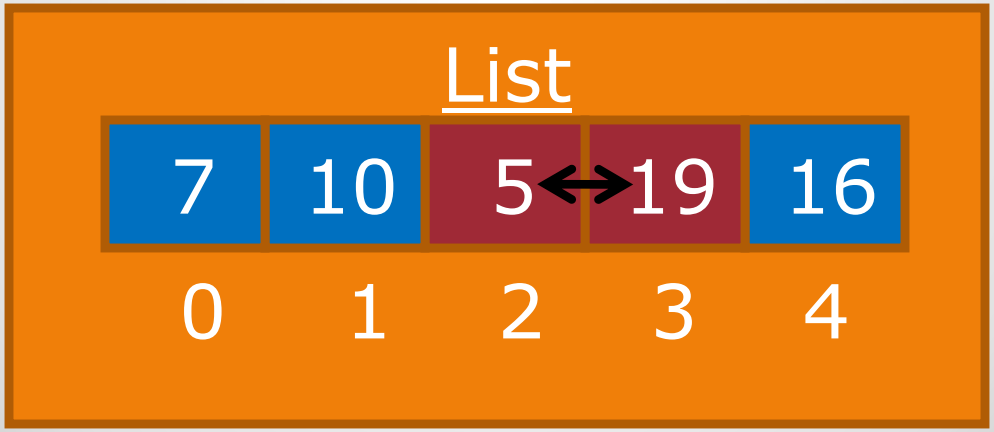
- Pass 1 (continued)
- Compare elements at index 1 and 2.
- These elements are in order so do NOT swap.



Bubble Sort

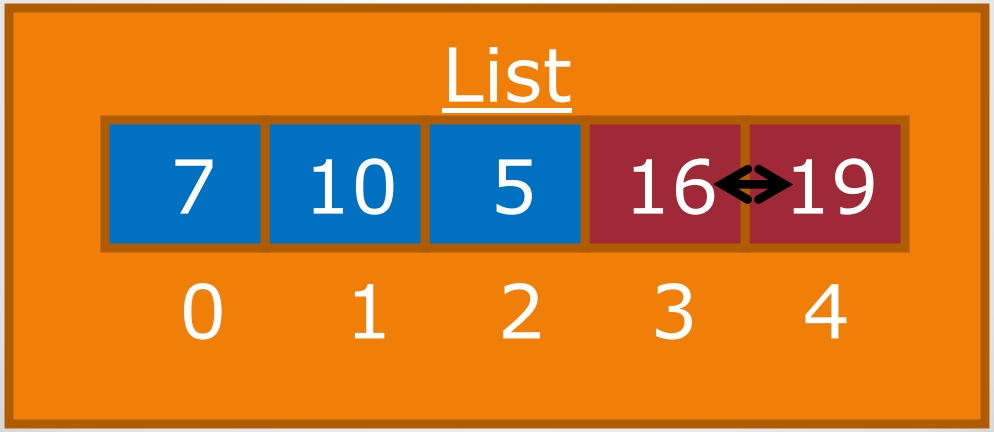
Bubble Sort

- Pass 1 (continued)
- Compare elements at index 2 and 3.
- These elements are out of order so swap.



Bubble Sort

- Pass 1 (continued)
- Compare elements at index 3 and 4.
- These elements are out of order so swap.
- **Pass 1 DONE**. This is what the array looks like after the first pass.
- Notice the high element is in its correct position.



- Pass 2
- Compare elements at index 0 and 1.
- No swap.

Bubble Sort

- Pass 2 (continued)
- Compare elements at index 1 and 2.
- Swap.

Bubble Sort

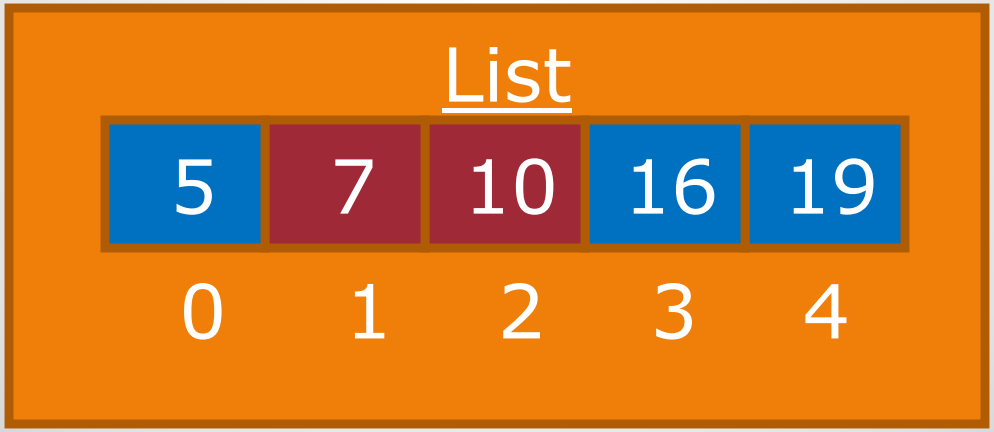
- Pass 2 (continued)
- Compare elements at index 2 and 3.
- No swap.
- **Pass 2 DONE**. This is what the array looks like after the second pass.
- Second highest element is in its correct position.

Bubble Sort

Bubble Sort

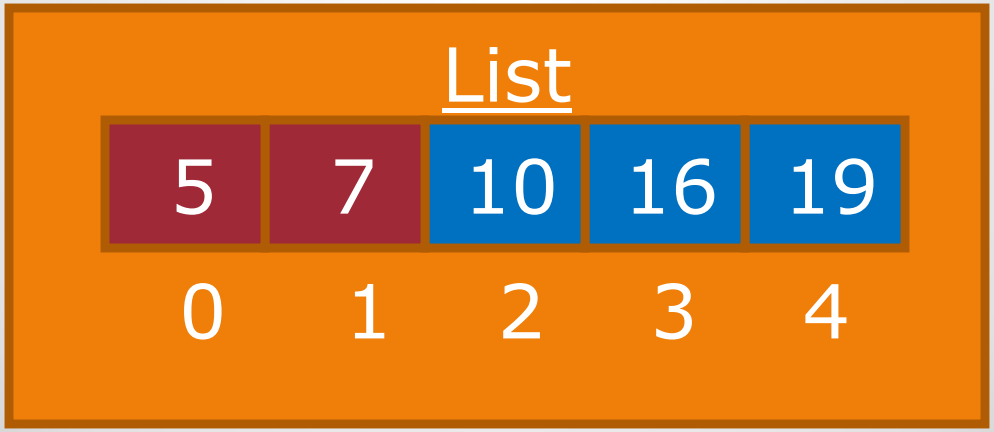
- Pass 3
- Compare elements at index 0 and 1.
- Swap.

- Pass 3 (continued)
- Compare elements at index 1 and 2.
- No swap.
- **Pass 3 DONE**. This is what the array looks like after the third pass.
- Third highest element is in its correct position.



Bubble Sort

- Pass 4
- Compare elements at index 0 and 1.
- No swap.
- This is what the array looks like after the fourth pass.
- **Pass 4 is DONE. The algorithm is finished.**



Bubble Sort

- All passes and swaps shown here…

Pass 1	10	7	19	5	16
	7	**10**	19	5	16
	7	**10**	**19**	5	16
	7	10	**5**	**19**	16
	7	10	5	**16**	**19**
Pass 2	**7**	**10**	5	16	19
	7	**5**	**10**	16	19
	7	5	**10**	**16**	19
Pass 3	**7**	**10**	5	16	19
	7	**5**	**10**	16	19
Pass 4	**5**	**7**	10	16	19
Done	5	7	10	16	19

Bold – Elements being compared.

Red – Swap was done

Blue – No swap

Green – Element in final place

All Passes and Swaps

Bubble Sort

```
void bubbleSort(int[] list)
   Declare int temp, iteration, index

   for iteration = 1 to (list.length – 1)

      for index = 0 to (list.length – iteration – 1)
         if list[index] > list[index + 1]
            temp = list[index]
            list[index] = list[index + 1]
            list[index + 1] = temp
         endIf
      endFor

   endFor
```

Outer loop is for the "passes"

This loop is one "pass"

Swap adjacent elements in the array that are out of order

- Now on to the analysis of bubble sort…

Bubble Sort Analysis

Sum of n Terms

- Let's add 10 Terms together!

10+9+8+7+6+5+4+3+2+1

- Pair the numbers up as follows:

10+9+8+7+6+5+4+3+2+1

There are 10 numbers (n=10).
There are 5 pairs (n/2 pairs).

Each pair adds up to 11.
5 * 11 = 55

- How many pairs? There are n numbers which means there are **n/2 pairs (5 in this case).**
- What is the sum of each pair? **n+1 is sum of pair (11 in this case)**
- **Total = (number of pairs) * (sum of each pair)**
- **Total = (n/2) * (n+1)**
- **Total** $= \frac{n(n+1)}{2}$

We are summing to n. The numerator of the formula requires that we multiply the number we are summing by one plus that number (n*(n+1)).

- Summation with odd number of terms (STILL WORKS).

5+4+3+2+1

- Pair the numbers up as follows:

5+4+3+2+1

There are 5 numbers (n=5).
There are 2.5 pairs (n/2 pairs).
Each pair adds up to 6.
6 * 2.5 = 15
The 3 in the middle is half of a pair.

- How many pairs? There are n numbers which means there are **n/2 pairs (2 in this case).**
- What is the sum of each pair? **n+1 is sum of pair (6 in this case)**
- **Total = (number of pairs) * (sum of each pair)**
- **Total = (n/2) * (n+1)**
- **Total** $= \frac{n(n+1)}{2}$

Sum of n Terms (Odd # of Terms)

Using the Sum Formula

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

Replace n with the ending number (6)

n = 6:

$$\frac{n(n+1)}{2} = \frac{6(6+1)}{2} = \frac{6(7)}{2} = \frac{42}{2} = 21$$

Replace n with the ending number (10)

n=10:

$$\frac{n(n+1)}{2} = \frac{10(10+1)}{2} = \frac{10(11)}{2} = \frac{110}{2} = 55$$

Replace n with the ending number (n-1). Wherever you see n replace it with n-1. This sums to one less than n.

n=(n-1):

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

- The number of comparisons done for each pass decreases for each pass.

Start	10	7	19	5	16
Pass 1	**7**	**10**	19	5	16
	7	**10**	**19**	5	16
	7	10	**5**	**19**	16
	7	10	5	**16**	**19**
Pass 2	**7**	**10**	5	16	19
	7	**5**	**10**	16	19
	7	5	**10**	**16**	19
Pass 3	**7**	**10**	5	16	19
	7	**5**	**10**	16	19
Pass 4	**5**	**7**	10	16	19
Done	5	7	10	16	19

Pass 1 → 4 comparisons

Pass 2 → 3 comparisons

Pass 3 → 2 comparisons

Pass 4 → 1 comparison

Bubble Sort Analysis

Bubble Sort Analysis

- The collection in this example has 5 elements so n=5.
- Here are the comparisons for each pass in terms of n:
  - Pass 1 → 4 comparisons is (n-1) comparisons
  - Pass 2 → 3 comparisons is (n-2) comparisons
  - Pass 3 → 2 comparisons is (n-3) comparisons
  - Pass 4 → 1 comparison is (n-4) comparisons
- How many pairs? (n-1)/2
- What is the sum of each pair? (n-1)+1 = n
- **Total = (number of pairs) * (sum of each pair)**
- **Total = ((n-1)/2) * (n)**
- **Total** $= \frac{(n-1)*n}{2}$
- **Total** $= \frac{(5-1)*5}{2}$
- **Total** $= \frac{4*5}{2}$
- **Total = 10**

WHY USE n(n-1) INSTEAD OF n(n+1) HERE?

We must sum up to n-1 (instead of n) because n-1 is the number of comparisons for pass 1. The numerator of the formula requires that we multiply the number we are summing to by one plus that number. In this case we are summing to n-1 (not n). So, we multiply (n-1)*n (n is one plus the number).

- There are $\frac{n(n-1)}{2}$ comparisons.

Simplify in terms of big O:

- $f(n) = \frac{n(n-1)}{2}$
- $= \frac{(n^2-n)}{2}$ ← Distribute n
- $= \frac{n^2}{2} - \frac{n}{2}$ ← Distribute 1/2
- $= n^2 - n$ ← Remove ½ constant from each term
- $= n^2$ ← Remove lower order terms

- **$f(n) \in O(n^2)$**

Bubble Sort – Big O

Bubble Sort (Normal)

- Runtimes for different cases of input data set.

- Average Data (randomized) is $O(n^2)$
- Sorted Data is $O(n^2)$

- **The algorithm as described (no optimizations) will always run the full nested loops so it will be $O(n^2)$ no matter what the order of the input data.**

- There is no best case of data that will speed up the plain algorithm as described so far.

Bubble Sort Analysis (Normal) – No Optimizations

- Runtimes for different cases of input data.

Input Data	$\Omega(g(n))$	$O(g(n))$	$\Theta(g(n))$
Average Case	n^2	n^2	n^2
Best Case (already sorted low to high)	n^2	n^2	n^2
Worst Case (sorted high to low)	n^2	n^2	n^2

- Best case input data does not speed up runtimes because the inner loop always runs all iterations.

Bubble Sort Analysis (Normal) – No Optimizations

Bubble Sort Optimization

- We can add an optimization to bubble sort to minimize the number of passes that are done.
- We must keep track of whether or not a swap is done during a pass. You can use a Boolean variable for this. For example, set swapDone=true if a swap is performed.
- If swapDone is false then it can immediately stop both loops, the collection is sorted (inner loop only runs once).

- Average Data (randomized) – $O(n^2)$
- **Sorted Data – O(n)**
- **If we use the optimization and the data is sorted, then the best-case runtime is O(n). It only needs to make one pass through the data.**

Bubble Sort Optimization

Bubble Sort Analysis with Optimization

- Runtimes with optimization for different cases of input data.

Input Data	$\Omega(g(n))$	$O(g(n))$	$\Theta(g(n))$
Average Case	n^2	n^2	n^2
Best Case (already sorted low to high)	**n**	**n**	**n**
Worst Case (sorted high to low)	n^2	n^2	n^2

- Best case input data speeds up runtime because inner loop does not run.

- **End of Slides**

End of Slides