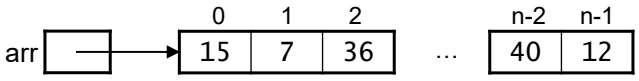
Unit 7, Part 1

A First Look at Sorting and Algorithm Analysis

Computer Science S-111
Harvard University

David G. Sullivan, Ph.D.

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
- Goal: minimize the number of **comparisons *C*** and the number of **moves *M*** needed to sort the array.
  - 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)`

Defining a Swap Method

- It would be helpful to have a method that swaps two elements of the array.
- Why won't the following work?

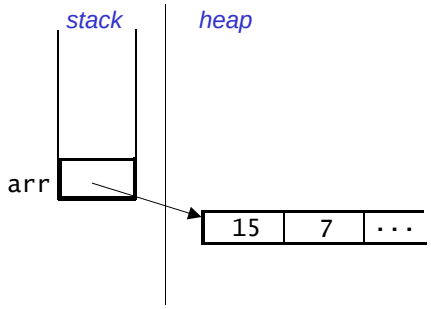
```
private static void swap(int a, int b) {
    int temp = a;
    a = b;
    b = temp;
}
```

An Incorrect Swap Method

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

- Trace through the following lines to see the problem:

```
int[] arr = {15, 7, …};
swap(arr[0], arr[1]);
```



A Correct Swap Method

- This method works:

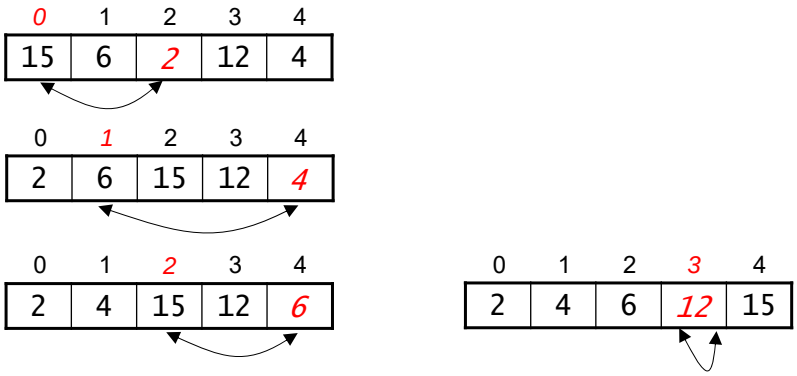
```
private static void swap(int[] arr, int a, int b) {
    int temp = arr[a];
    arr[a] = arr[b];
    arr[b] = temp;
}
```

- Trace through the following with a memory diagram to convince yourself that it works:

```
int[] arr = {15, 7, …};
swap(arr, 0, 1);
```

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:



Why don't we need to consider position 4?

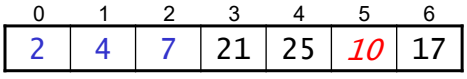
Selecting an Element

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

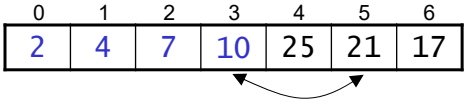
example for `i = 3`:

- 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

`indexMin: 3, 5`



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



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;
}
```

- 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

- Some algorithms are much more efficient than others.
- The *time efficiency* or *time complexity* of an algorithm is some measure of the number of operations that it performs.
  - for sorting, we'll focus on comparisons and moves
- We want to characterize how the number of operations depends on the size, n, of the input to the algorithm.
  - for sorting, n is the length of the array
  - how does the number of operations grow as n grows?
- We'll express the number of operations as functions of n
  - `C(n)` = number of comparisons for an array of length n
  - `M(n)` = number of moves for an array of length n

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 ____ comparisons to find `indexSmallest`
  on 2nd pass, it performs ____ comparisons
  …
  on the (n–1)st pass, it performs 1 comparison

- Adding them up: $C(n) = 1 + 2 + \ldots + (n - 2) + (n - 1)$

Counting Comparisons by Selection Sort (cont.)

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

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

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

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

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

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

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

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

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

Focusing on the Largest Term

- When n is large, mathematical expressions of n are dominated by their "largest" term — i.e., the term that grows fastest as a function of n.
  - example:

n	$n^2/2$	$n/2$	$n^2/2 - n/2$
10	50	5	45
100	5000	50	4950
10000	50,000,000	5000	49,995,000

- In characterizing the time complexity of an algorithm, we'll focus on the largest term in its operation-count expression.
  - for selection sort, $c(n) = n^2/2 - n/2 \approx n^2/2$
- In addition, we'll typically ignore the coefficient of the largest term (e.g., $n^2/2 \rightarrow n^2$).

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 $O(n^2)$
- Common classes of algorithms:

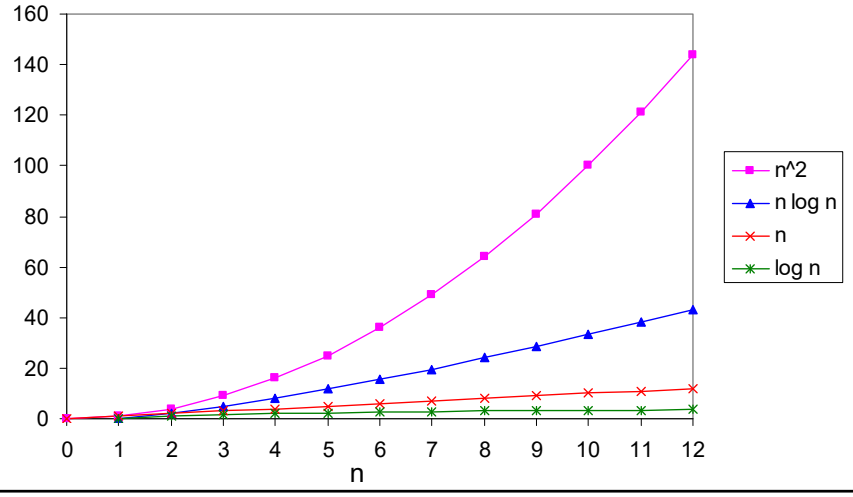
name	example expressions	big-O notation
constant time	$1, 7, 10$	$O(1)$
logarithmic time	$3\log_{10}n, \log_2 n + 5$	$O(\log n)$
linear time	$5n, 10n - 2\log_2 n$	$O(n)$
nlogn time	$4n\log_2 n, n\log_2 n + n$	$O(n \log n)$
quadratic time	$2n^2 + 3n, n^2 - 1$	$O(n^2)$
exponential time	$2^n, 5e^n + 2n^2$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

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

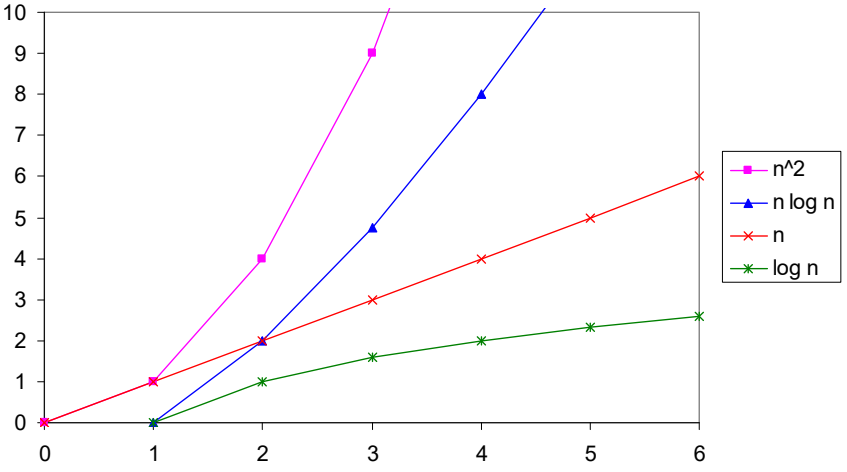
Ordering of Functions

- 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 $n >= 1$

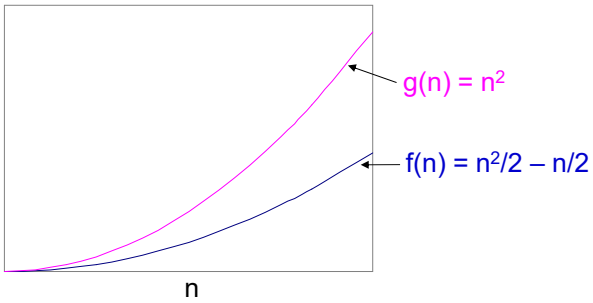


Big-*O* Time Analysis of Selection Sort

- 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): ?

Mathematical Definition of Big-*O* Notation

- $f(n) = O(g(n))$ if there exist positive constants c and n_0 such that $f(n) <= cg(n)$ for all $n >= n_0$
- Example: $f(n) = n^2/2 - n/2$ is $O(n^2)$, because
  $n^2/2 - n/2 <= n^2$ for all $n >= 0$.
  $c = 1$ $n_0 = 0$



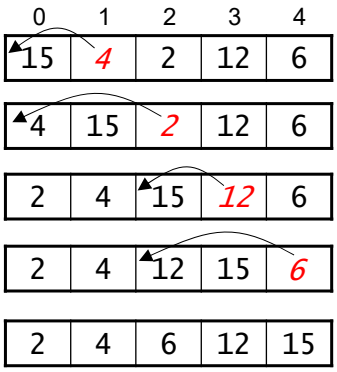
- Big-*O* notation specifies an *upper bound* on a function f(n) as n grows large.

Big-*O* Notation and Tight Bounds

- Strictly speaking, big-O notation provides an upper bound, *not* a tight bound (upper and lower).
- Example:
  - $3n - 3$ is $O(n^2)$ because $3n - 3 <= n^2$ for all $n >= 1$
  - $3n - 3$ is also $O(2^n)$ because $3n - 3 <= 2^n$ for all $n >= 1$
- However, it is common to use big-O notation to characterize a function as closely as possible – as if it specified a tight bound.
  - for our example, we would say that $3n - 3$ is $O(n)$
  - this is how you should use big-O in this class!

Insertion Sort

- Basic idea:
  - going from left to right, "insert" each element into its proper place with respect to the elements to its left
  - "slide 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
  - …

Inserting an Element

- 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

12	5	2	13	18	4

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
  - each element is only compared to the element to its left
  - we never execute the do-while loop!
  - `C(n)` =_____, `M(n)` = _____, running time = _____

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

Shell Sort

- Developed by Donald Shell
- Improves on insertion sort
  - takes advantage of the fact that it's fast for almost-sorted arrays
  - eliminates a key disadvantage: an element may need to move many times to get to where it belongs.
- Example: if the largest element starts out at the beginning of the array, it moves one place to the right on *every* insertion!

0	1	2	3	4	5	…	1000
999	42	56	30	18	23	…	11

- Shell sort uses larger moves that allow elements to quickly get close to where they belong in the sorted array.

Sorting Subarrays

- Basic idea:
  - use insertion sort on subarrays that contain elements separated by some increment `incr`
    - increments allow the data items to make larger "jumps"
  - repeat using a decreasing sequence of increments
- Example for an initial increment of 3:

0	1	2	3	4	5	6	7
36	18	10	27	3	20	9	8

  - three subarrays:
    1) elements 0, 3, 6 2) elements 1, 4, 7 3) elements 2 and 5
- Sort the subarrays using insertion sort to get the following:

0	1	2	3	4	5	6	7
9	3	10	27	8	20	36	18

- Next, we complete the process using an increment of 1.

Shell Sort: A Single Pass

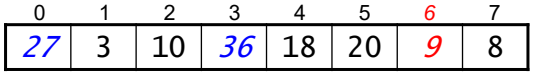
- We *don't* actually consider the subarrays one at a time.
- For each element from position `incr` to the end of the array, we insert the element into its proper place with respect to the elements *from its subarray* that come before it.
- The same example (`incr` = 3):

0	1	2	3	4	5	6	7
36	18	10	*27*	3	20	9	8
27	*18*	10	36	*3*	20	9	8
27	3	*10*	36	18	*20*	9	8
27	3	10	*36*	18	20	*9*	8
9	*3*	10	27	*18*	20	36	*8*
9	3	10	27	8	20	36	18

Inserting an Element in a Subarray

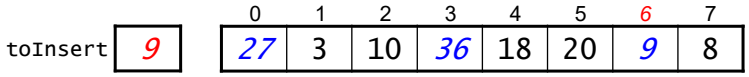
- When we consider element `i`, the other elements in its subarray are already sorted with respect to each other.

example for `i = 6`:
`(incr = 3)`

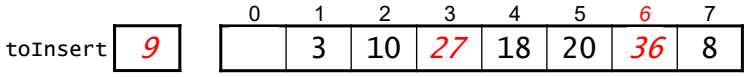


the other element's in 9's subarray (the 27 and 36) are already sorted with respect to each other

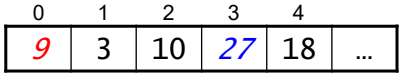
- To insert element `i`:
  - make a copy of element `i`, storing it in the variable `toInsert`:



  - consider elements `i-incr`, `i-(2*incr)`, `i-(3*incr)`,…
    - if an element `>` `toInsert`, slide it right *within the subarray*
    - stop at the first element `<=` `toInsert`



  - copy `toInsert` into the "hole":



The Sequence of Increments

- Different sequences of decreasing increments can be used.
- Our version uses values that are one less than a power of two.
  - $2^k - 1$ for some k
  - … 63, 31, 15, 7, 3, 1
  - can get to the next lower increment using integer division:

```
incr = incr/2;
```

- Should avoid numbers that are multiples of each other.
  - otherwise, elements that are sorted with respect to each other in one pass are grouped together again in subsequent passes
    - repeat comparisons unnecessarily
    - get fewer of the large jumps that speed up later passes
  - example of a bad sequence: 64, 32, 16, 8, 4, 2, 1
    - what happens if the largest values are all in odd positions?

Implementation of Shell Sort

```
public static void shellSort(int[] arr) {
    int incr = 1;
    while (2 * incr <= arr.length) {
        incr = 2 * incr;
    }
    incr = incr - 1;

    while (incr >= 1) {
        for (int i = incr; i < arr.length; i++) {
            if (arr[i] < arr[i-incr]) {
                int toInsert = arr[i];

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

                arr[j] = toInsert;
            }
        }
        incr = incr/2;
    }
}
```

(If you replace `incr` with `1` in the for-loop, you get the code for insertion sort.)

Time Analysis of Shell Sort

- Difficult to analyze precisely
  - typically use experiments to measure its efficiency
- With a bad interval sequence, it's $O(n^2)$ in the worst case.
- With a good interval sequence, it's better than $O(n^2)$.
  - at least $O(n^{1.5})$ in the average and worst case
  - some experiments have shown average-case running times of $O(n^{1.25})$ or even $O(n^{7/6})$
- Significantly better than insertion or selection for large n:

n	n^2	$n^{1.5}$	$n^{1.25}$
10	100	31.6	17.8
100	10,000	1000	316
10,000	100,000,000	1,000,000	100,000
10^6	10^{12}	10^9	3.16×10^7

- We've wrapped insertion sort in another loop and increased its efficiency! The key is in the larger jumps that Shell sort allows.

Practicing Time Analysis

- Consider the following static method:

```
public static int mystery(int n) {
    int x = 0;
    for (int i = 0; i < n; i++) {
        x += i;          // statement 1
        for (int j = 0; j < i; j++) {
            x += j;
        }
    }
    return x;
}
```

- What is the big-O expression for the number of times that statement 1 is executed as a function of the input n?

What about now?

- Consider the following static method:

```
public static int mystery(int n) {
    int x = 0;
    for (int i = 0; i < 3*n + 4; i++) {
        x += i;          // statement 1
        for (int j = 0; j < i; j++) {
            x += j;
        }
    }
    return x;
}
```

- What is the big-O expression for the number of times that statement 1 is executed as a function of the input n?

Practicing Time Analysis

- Consider the following static method:

```
public static int mystery(int n) {
    int x = 0;
    for (int i = 0; i < n; i++) {
        x += i;          // statement 1
        for (int j = 0; j < i; j++) {
            x += j;      // statement 2
        }
    }
    return x;
}
```

- What is the big-O expression for the number of times that statement 2 is executed as a function of the input n?

value of i	number of times statement 2 is executed

Bubble Sort

- Perform a sequence of passes from left to right
  - each pass swaps 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
  - we don't need to consider them in subsequent passes.
- Example:

	0	1	2	3	4
	28	24	37	15	5
after the first pass:	24	28	15	5	*37*
after the second:	24	15	5	*28*	*37*
after the third:	15	5	*24*	*28*	*37*
after the fourth:	5	*15*	*24*	*28*	*37*

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);
                }
            }
        }
    }
}
```

- Nested loops:
  - the inner loop performs a single pass
  - the outer loop governs:
    - the number of passes (`arr.length - 1`)
    - the ending point of each pass (the current value of `i`)

Time Analysis of Bubble Sort

- Comparisons (n = length of array):
  - they are performed in the inner loop
  - *how many repetitions does each execution of the inner loop perform?*

value of i	number of comparisons
n – 1	n – 1
n – 2	n – 2
…	…
2	2
1	1

1 + 2 + … + n – 1 =

```
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);
            }
        }
    }
}
```

Time Analysis of Bubble Sort

- Comparisons: the kth pass performs 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:
    - $M(n) =$
  - in the best case:

- Running time:
  - $C(n)$ 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.
  - can't do much worse than bubble!