

Lesson 83: Sorting #2 (W29D3)

Balboa High School

Michael Ferraro

March 16, 2016

Students will learn about Mergesort, Quicksort, and about the relative efficiencies of the sorting algorithms.

Do Now

On **paper**, perform Selection Sort and Insertion Sort on the array below, with a partner, keeping track of the following:

	# comparisons	# swaps/moves	total steps
Selection Sort			
Insertion Sort			

26	-4	18	19	20	0
----	----	----	----	----	---

Mergesort

- recursive!

Mergesort

- recursive!
- two steps:
 - split the list in \approx half, recursively sorting left and right halves
 - merge the two sorted halves back together

Mergesort

- recursive!
- two steps:
 - split the list in \approx half, recursively sorting left and right halves
 - merge the two sorted halves back together
- base cases:
 - only one elt? already sorted
 - two elts? if not already in order, swap elts

Mergesort

- recursive!
- two steps:
 - split the list in \approx half, recursively sorting left and right halves
 - merge the two sorted halves back together
- base cases:
 - only one elt? already sorted
 - two elts? if not already in order, swap elts
- online examples to view later:
 - [cool applet](#)
 - [sorting-algorithms.com](#)

Let's see a demonstration...

Mergesort

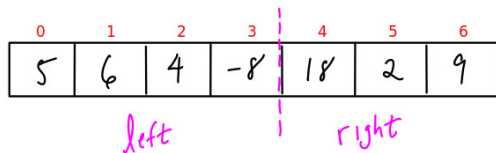
0	1	2	3	4	5	6
5	6	4	-8	18	2	9

Base Cases:

1. Single elt? Do nothing!
2. Two elts? Swap, if necessary.

STACK:

Mergesort



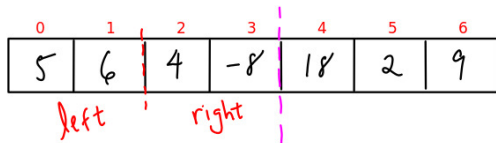
Base Cases:

1. Single elt? Do nothing!
2. Two elts? Swap, if necessary.

STACK:

mS(0,3) left
mS(4,6) right
merge[(0,3)&(4,6)]
mS(0,6)

Mergesort



Base Cases:

1. Single elt? Do nothing!
2. Two elts? Swap, if necessary.

STACK:

→ $ms(0,1)$ left

→ $ms(2,3)$ right

→ $merge[0,1] \& [2,3]$

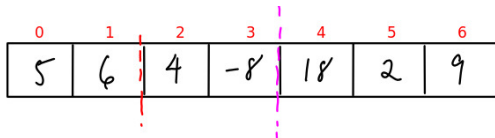
→ $ms(0,3)$

$ms(4,6)$

$merge[0,3] \& [4,6]$

$ms(0,6)$

Mergesort

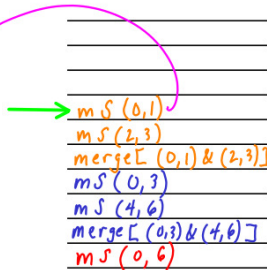


Base Cases:

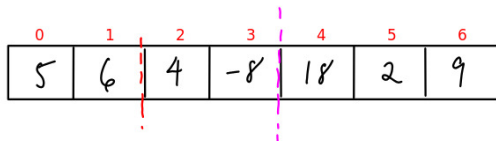
1. Single elt? Do nothing!
2. Two elts? Swap, if necessary.

Already in order.

STACK:



Mergesort



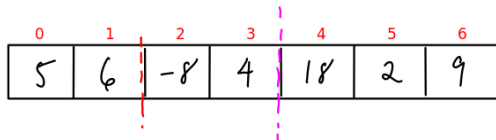
Base Cases:

1. Single elt? Do nothing!
2. Two elts? Swap, if necessary.

STACK:

~~mS(0,1)~~
mS(2,3)
merge[(0,1) & (2,3)]
mS(0,3)
mS(4,6)
merge[(0,3) & (4,6)]
mS(0,6)

Mergesort



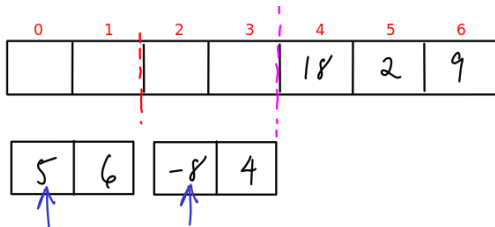
Base Cases:

1. Single elt? Do nothing!
2. Two elts? Swap, if necessary.

STACK:

~~ms(0,1)~~
~~ms(2,3)~~
→ merge[(0,1) & (2,3)]
ms(0,3)
ms(4,6)
merge[(0,3) & (4,6)]
ms(0,6)

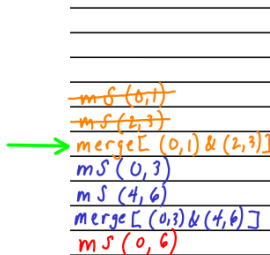
Mergesort



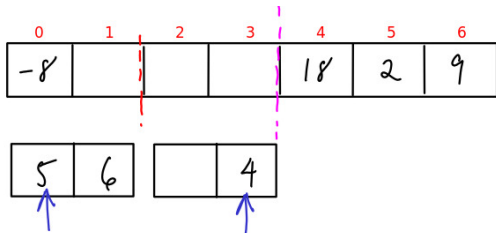
Base Cases:

1. Single elt? Do nothing!
2. Two elts? Swap, if necessary.

STACK:



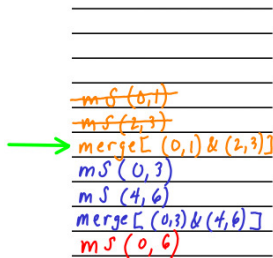
Mergesort



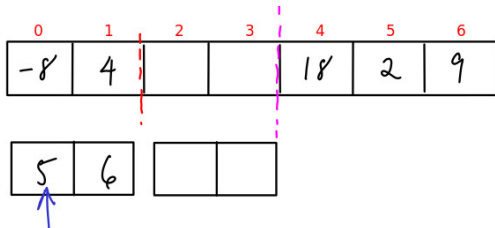
Base Cases:

1. Single elt? Do nothing!
2. Two elts? Swap, if necessary.

STACK:



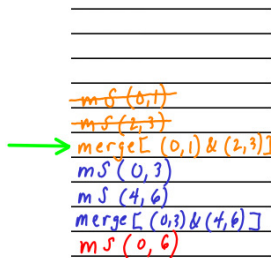
Mergesort



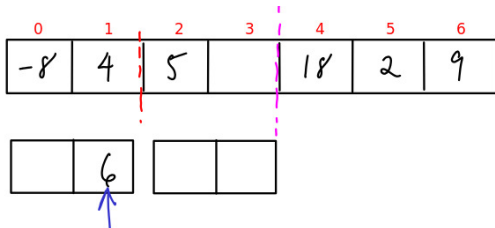
Base Cases:

1. Single elt? Do nothing!
2. Two elts? Swap, if necessary.

STACK:



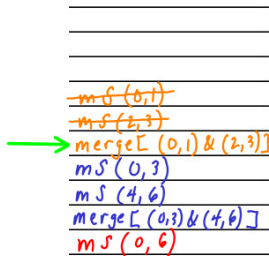
Mergesort



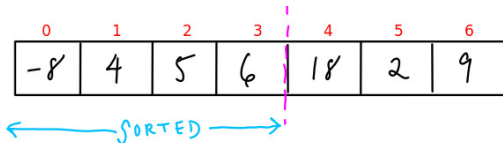
Base Cases:

1. Single elt? Do nothing!
2. Two elts? Swap, if necessary.

STACK:



Mergesort



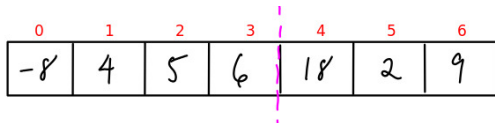
Base Cases:

1. Single elt? Do nothing!
2. Two elts? Swap, if necessary.

STACK:

~~$mS(0,1)$~~
 ~~$mS(2,3)$~~
 ~~$merge[0,1] \& [2,3]$~~
→ $mS(0,3)$
 $mS(4,6)$
 $merge[0,3] \& [4,6]$
 $mS(0,6)$

Mergesort



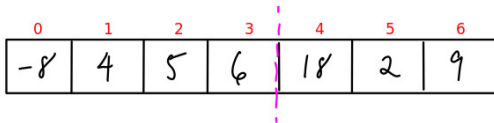
Base Cases:

1. Single elt? Do nothing!
2. Two elts? Swap, if necessary.

STACK:

 ~~$ms(0,1)$~~
 ~~$ms(2,3)$~~
 ~~$merge[(0,1) \& (2,3)]$~~
 ~~$ms(0,3)$~~
→ $ms(4,6)$
 $merge[(0,3) \& (4,6)]$
 $ms(0,6)$

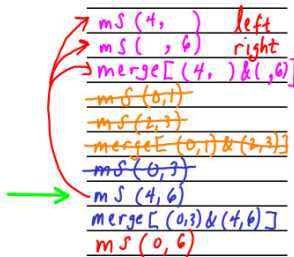
Mergesort



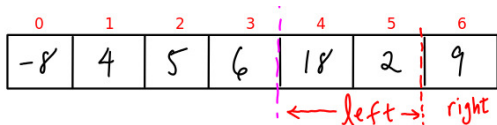
Base Cases:

1. Single elt? Do nothing!
2. Two elts? Swap, if necessary.

STACK:



Mergesort



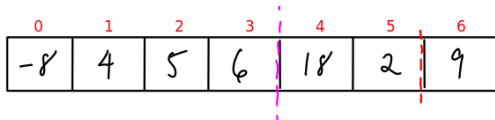
Base Cases:

1. Single elt? Do nothing!
2. Two elts? Swap, if necessary.

STACK:

→ $ms(4, 5)$ left
 $ms(6, 6)$ right
 $merge[(4, 5) \& (6, 6)]$
 ~~$ms(0, 1)$~~
 ~~$ms(2, 3)$~~
 ~~$merge[(0, 1) \& (2, 3)]$~~
 ~~$ms(0, 3)$~~
 $ms(4, 6)$
 $merge[(0, 3) \& (4, 6)]$
 $ms(0, 6)$

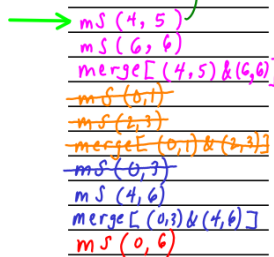
Mergesort



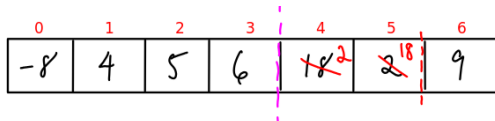
Base Cases:

1. Single elt? Do nothing!
2. Two elts? Swap, if necessary.

STACK:



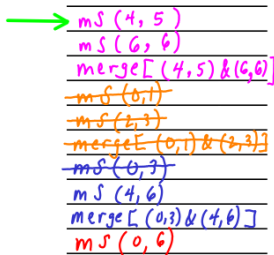
Mergesort



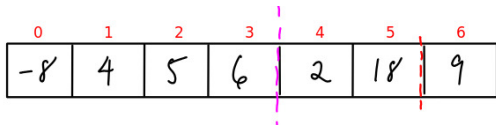
Base Cases:

1. Single elt? Do nothing!
2. Two elts? Swap, if necessary.

STACK:



Mergesort



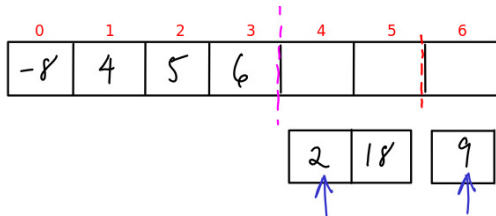
Base Cases:

1. Single elt? Do nothing!
2. Two elts? Swap, if necessary.

STACK:

~~ms(4, 5)~~
ms(6, 6)
merge[(4, 5) & (6, 6)]
~~ms(0, 1)~~
~~ms(2, 3)~~
~~merge[(0, 1) & (2, 3)]~~
~~ms(0, 3)~~
ms(4, 6)
merge[(0, 3) & (4, 6)]
ms(0, 6)

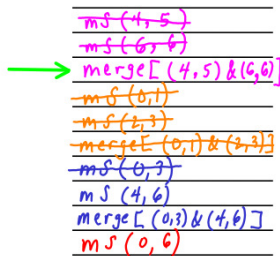
Mergesort



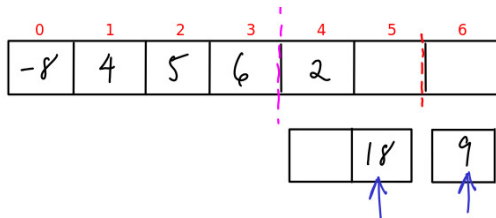
Base Cases:

1. Single elt? Do nothing!
2. Two elts? Swap, if necessary.

STACK:



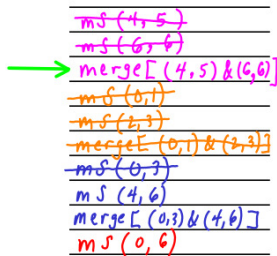
Mergesort



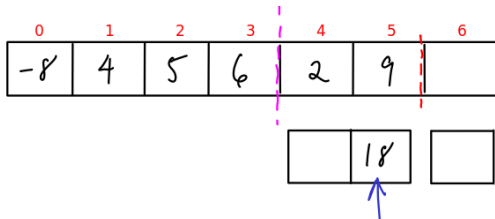
Base Cases:

1. Single elt? Do nothing!
2. Two elts? Swap, if necessary.

STACK:



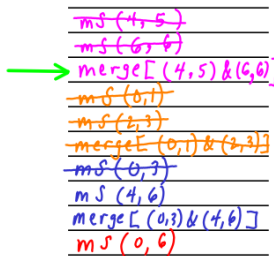
Mergesort



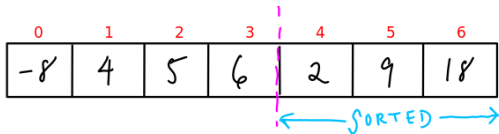
Base Cases:

1. Single elt? Do nothing!
2. Two elts? Swap, if necessary.

STACK:



Mergesort



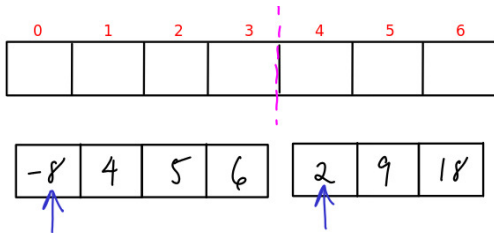
Base Cases:

1. Single elt? Do nothing!
2. Two elts? Swap, if necessary.

STACK:

~~$ms(4, 5)$~~
 ~~$ms(6, 6)$~~
 ~~$merge[4, 5] \& [6, 6]$~~
 ~~$ms(0, 1)$~~
 ~~$ms(2, 3)$~~
 ~~$merge[0, 1] \& [2, 3]$~~
 ~~$ms(0, 3)$~~
 $ms(4, 6)$
 $merge[0, 3] \& [4, 6]$
 $ms(0, 6)$

Mergesort



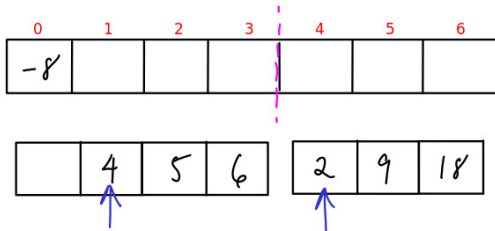
Base Cases:

1. Single elt? Do nothing!
2. Two elts? Swap, if necessary.

STACK:

~~ms(4,5)~~
~~ms(6,6)~~
~~mergeL(4,5)&(6,6)]~~
~~ms(0,1)~~
~~ms(2,3)~~
~~mergeL(0,1)&(2,3)]~~
~~ms(0,3)~~
~~ms(4,6)~~
→ mergeL(0,3)&(4,6)]
ms(0,6)

Mergesort



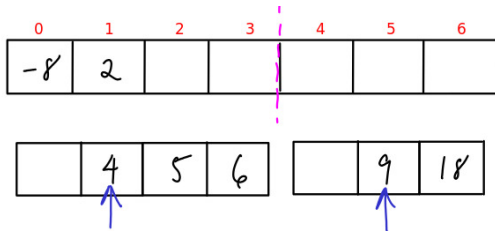
Base Cases:

1. Single elt? Do nothing!
2. Two elts? Swap, if necessary.

STACK:

~~ms(4,5)~~
~~ms(6,6)~~
~~mergeE(4,5)&(6,6)]~~
~~ms(0,1)~~
~~ms(2,3)~~
~~mergeE(0,1)&(2,3)]~~
~~ms(0,3)~~
~~ms(4,6)~~
→ mergeE(0,3)&(4,6)]
ms(0,6)

Mergesort



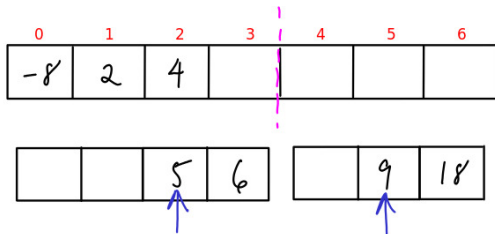
Base Cases:

1. Single elt? Do nothing!
2. Two elts? Swap, if necessary.

STACK:

~~ms(4,5)~~
~~ms(6,6)~~
~~mergeE(4,5)&(6,6)]~~
~~ms(0,1)~~
~~ms(2,3)~~
~~mergeE(0,1)&(2,3)]~~
~~ms(0,3)~~
~~ms(4,6)~~
→ mergeE(0,3)&(4,6)]
ms(0,6)

Mergesort



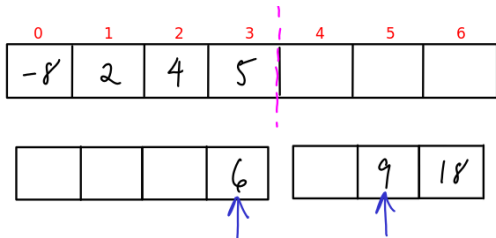
Base Cases:

1. Single elt? Do nothing!
2. Two elts? Swap, if necessary.

STACK:

~~ms(4,5)~~
~~ms(6,6)~~
~~merge[(4,5)&(6,6)]~~
~~ms(0,1)~~
~~ms(2,3)~~
~~merge[(0,1)&(2,3)]~~
~~ms(0,3)~~
~~ms(4,6)~~
→ merge[(0,3)&(4,6)]
~~ms(0,6)~~

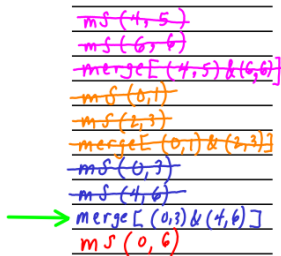
Mergesort



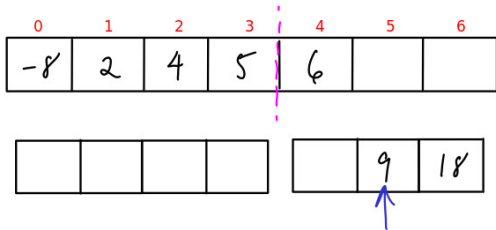
Base Cases:

1. Single elt? Do nothing!
2. Two elts? Swap, if necessary.

STACK:



Mergesort



Base Cases:

1. Single elt? Do nothing!
2. Two elts? Swap, if necessary.

STACK:

~~ms(4,5)~~
~~ms(6,6)~~
~~merge[(4,5)&(6,6)]~~
~~ms(0,1)~~
~~ms(2,3)~~
~~merge[(0,1)&(2,3)]~~
~~ms(0,3)~~
~~ms(4,6)~~
→ merge[(0,3)&(4,6)]
ms(0,6)

Mergesort



Base Cases:

1. Single elt? Do nothing!
2. Two elts? Swap, if necessary.

STACK:

~~ms(4, 5)~~
~~ms(6, 6)~~
~~mergeE(4, 5) & (6, 6)]~~
~~ms(0, 1)~~
~~ms(2, 3)~~
~~mergeE(0, 1) & (2, 3)]~~
~~ms(0, 3)~~
~~ms(4, 6)~~
~~mergeE(0, 3) & (4, 6)]~~
→ ✓ ms(0, 6)

Mergesort

Try Mergesort on the Do Now array, keeping track of the total number of comparisons, swaps, moves (i.e., # of steps):

26	-4	18	19	20	0
----	----	----	----	----	---

Let's see how Mergesort performs relative to Selection Sort and Insertion Sort.

Mergesort

Let's tour briefly through [Mergesort.java](#) so you can map the Java code you see to the sort as you understand it...

Relative Efficiencies of Our Sorts

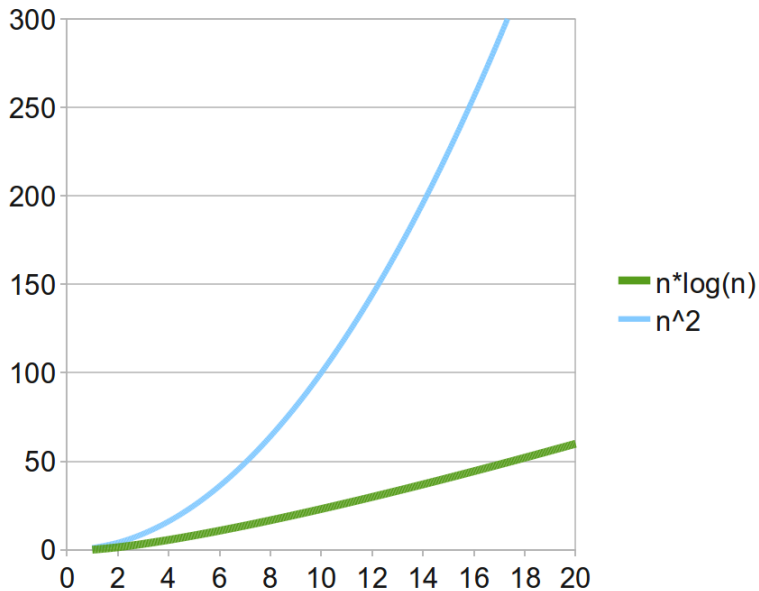
- When a programmer works with large data sets (e.g, Facebook, Google, or Twitter), choosing a more efficient algorithm may mean the difference between a winning product and one that nobody will use
- You can do *empirical* testing: unleash competing algorithms on a real data set and see which one consistently wins; sometimes this is a good approach
- What if you need to decide on an algorithm to use before you have real data? You need to do a theoretical analysis (discrete mathematics)
- One way to compare algorithms is to figure out how bad they'll be in the worst case (i.e., Insertion Sort when elts are in descending order, every elt gets shuffled)

Big-Oh Notation

- One way to express worst-case running time is to write the number of steps in terms of n in this way: $O(n^2)$
- Here are common Big-Oh values:

$O(1)$	regardless of # of elts, algorithm takes constant time (steps)
$O(n)$	as # of elts grows, the # of steps grows \approx proportionally
$O(n^2)$	n doubles \rightarrow steps \approx quadruple ($\times 2^2$), n triples \rightarrow steps $\approx \times 9$ ($\times 3^2$); e.g., simple multiplication of two n -digit numbers
$O(n \cdot \log n)$	grows at a rate greater than n , but far less than n^2

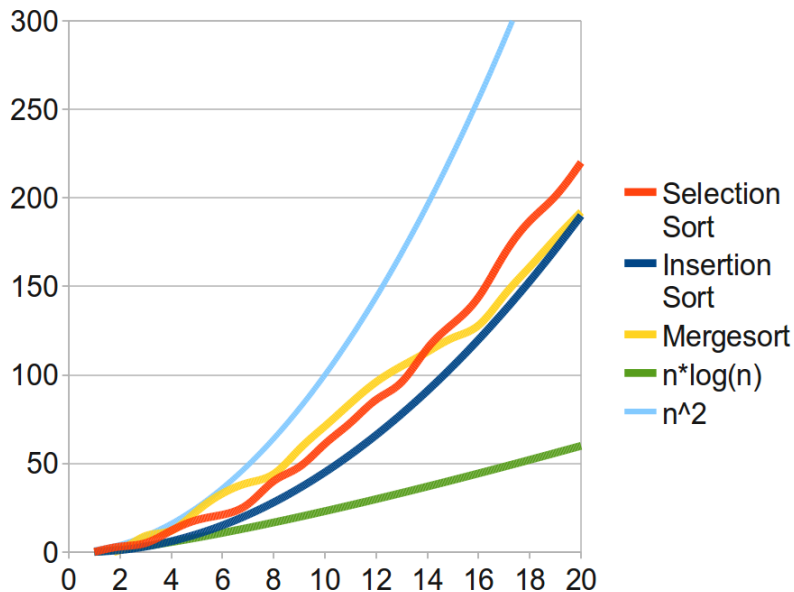
$O(n^2)$ vs. $O(n \cdot \log n)$



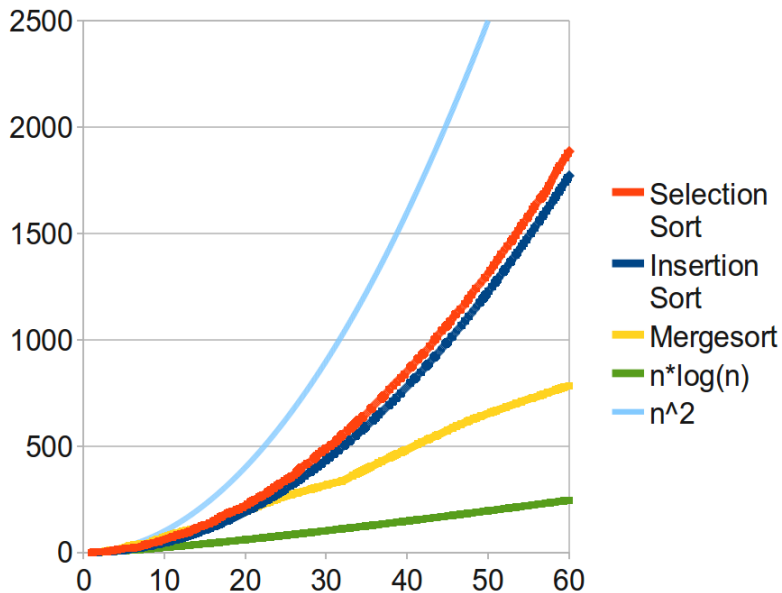
How do our sorts perform?

- In L82's `SortDriver.java`, find the commented-out block of code in `main()` and enable it.
- Watch as I demonstrate graphing the data...

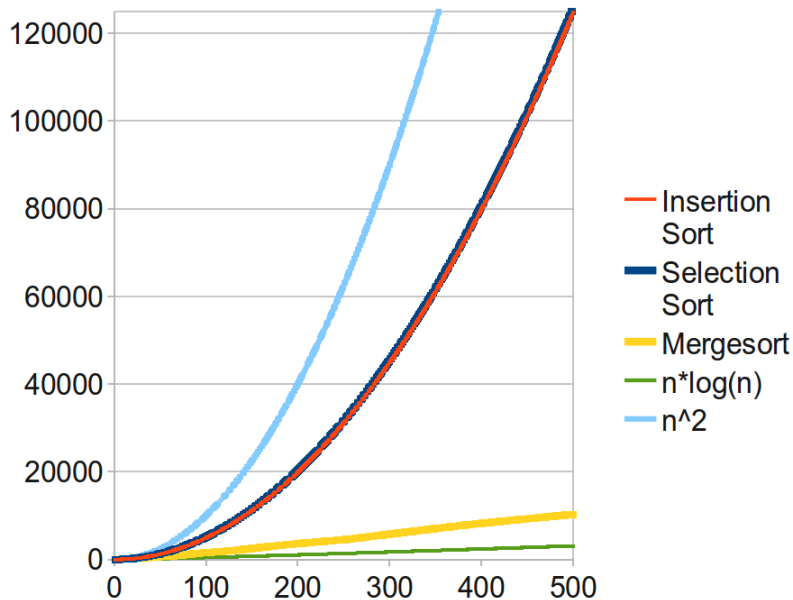
Up to 20 elts ($1 \leq n \leq 20$)



Up to 60 elts ($1 \leq n \leq 60$)



Up to 500 elts ($1 \leq n \leq 500$)



- Find the source code provided in Litvin §14.8
- Basic idea:
 - swap elts around a “pivot point” so that lesser elts end up on left and greater ones on right, sometimes necessary to move the pivot to a better location
 - apply recursively to elts to left & elts to right of pivot
 - pivot acts to *partition* elts into two halves — so by recursively attacking each half, you’re effectively “dividing and conquering” the problem
- Here’s a [demo](#)

- Quiz next class!
- Go through the lesson slides for the past two lessons and practice using the sorting algorithms.