2023-01-18

7.6.2
Insertion Sort

https://xkcd.com/1185/

Preconditions, Postconditions, and Invariants

- We learned about these already:
  - A **precondition** to a method must be true before entering the method.
  - A **postcondition** to a method must be true when leaving the method.
- An **invariant** is some condition that must always be true.
  - Example: In the class StudentDirectory, the data in the ArrayList "students" is always in sorted order by name.
- Preconditions, postconditions, and invariants are used to formally prove the **correctness** of algorithms.

Loop Invariants

A **loop invariant** is a condition that must be true at the beginning and end of the body of a loop. (It might not be true while the loop body is doing its work, like swaps.)

```java
// Precondition: values must be non-empty.
// Postcondition: The minimum value in "values" is returned.
public int minValue(int[] values) {
  int minResult = values[0];
  for (int i=1, n=values.length; i<n; i++) {
    // Loop invariant: minResult contains the minimum value in a[0]..a[i-1]
    if (values[i] < minResult) {
      minResult = values[i];
    }
  }
  return minResult;
}
```

What's the loop invariant in Selection Sort?

```java
// Postcondition: values will be in sorted ascending order
public void selectionSort(double[] values) {
  for (int i=0, n=values.length; i<n; i++) {
    int jMin = i;
    for (int j=i+1; j<n; j++) {
      if (values[j] < values[jMin]) {
        jMin = j;
      }
    }
    double temp = values[i];
    values[i] = values[jMin];
    values[jMin] = temp;
  }
}
```

Trick question: There were two loop invariants

```
// Postcondition: values will be in sorted ascending order
public void selectionSort(double[] values) {
  for (int i=0, n=values.length; i<n; i++) {
    // Loop invariant: values[0..i-1] is i smallest values, in sorted order
    int jMin = i;
    for (int j=i+1; j<n; j++) {
      // Loop invariant: values[jMin] is smallest value in values[i..j-1]
      if (values[j] < values[jMin]) {
        jMin = j;
      }
    }
    double temp = values[i];
    values[i] = values[jMin];
    values[jMin] = temp;
  }
}
```

Introducing Insertion Sort

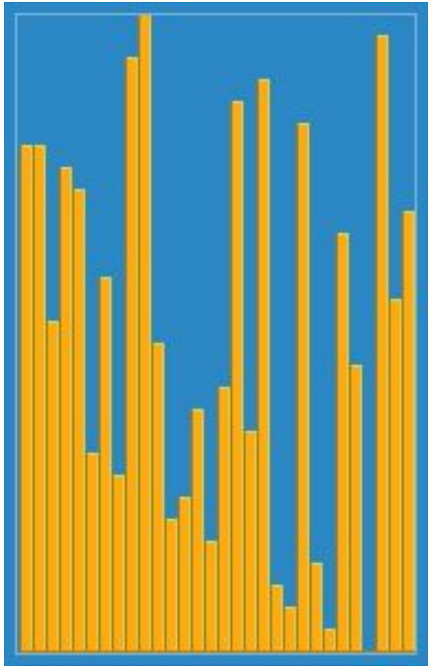
Insertion Sort is more complex than Selection Sort, but is much faster when the data is partially sorted.
Unfortunately, it is still a quadratic algorithm, that is, O(N^2).

Insertion Sort

Class	Sorting algorithm
Data structure	Array
Worst-case performance	$O(n^2)$ comparisons and swaps
Best-case performance	$O(n)$ comparisons, $O(1)$ swaps
Average performance	$O(n^2)$ comparisons and swaps
Worst-case space complexity	$O(n)$ total, $O(1)$ auxiliary

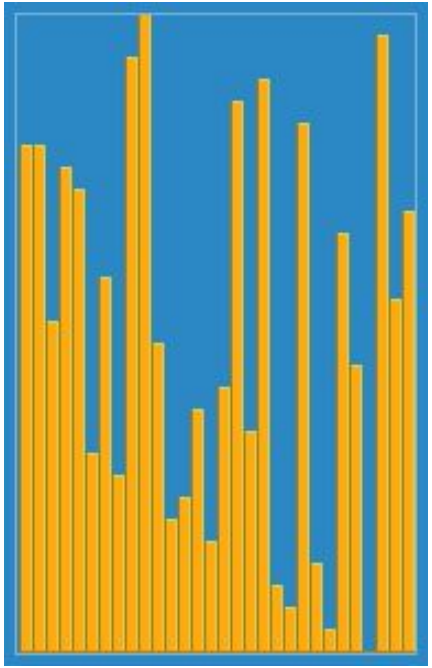
Selection sort

Class	Sorting algorithm
Data structure	Array
Worst-case performance	$O(n^2)$ comparisons, $O(n)$ swaps
Best-case performance	$O(n^2)$ comparisons, $O(1)$ swap
Average performance	$O(n^2)$ comparisons, $O(n)$ swaps
Worst-case space complexity	$O(1)$ auxiliary



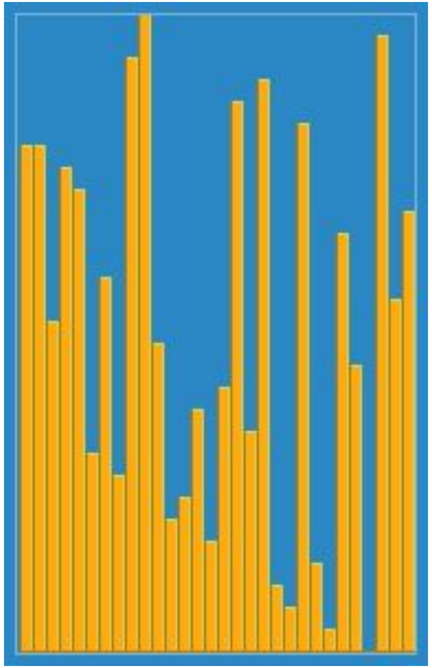
Insertion Sort Algorithm

- Insertion Sort has a loop invariant that for index i, **the entire sub-array to the left of i is in sorted order.**
  - This is slightly different from Selection Sort's loop invariant... how?
- Insertion Sort's outer loop starts with i = 1, that is, pointing to the **second** element in the array.
- Why? The sub-array to the left of i = 1, a[0..0], is in sorted order, because a one-element array is always in sorted order!



Insertion Sort Algorithm

```
// Note we start with second element i = 1
for i ← 1 to length(array)
    // Loop invariant: The array to the left of i is a sorted sub-array.
    // j will represent the insertion point of the value.
    j ← i
    while j > 0 and A[j-1] > A[j]
        // As we scan for insertion point, we swap the element
        // we are inserting (which will move to the left) with the
        // element to its left.
        swap A[j], A[j-1]
        j ← j - 1
    end while
end for
```



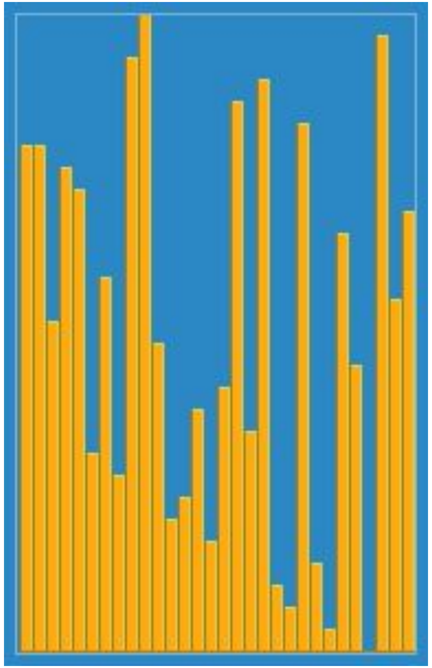
Insertion Sort Algorithm: Swapless Edition

```
// A bit more efficient than swapping. Less read/write operations.
// Note we start with second element i = 1
for i ← 1 to length(array)
        // Loop invariant: The array to the left of i is a sorted sub-array.

        // Save the value at A[i], as it may be overwritten
        x ← A[i]

        // j will represent the insertion point of the value.
        j ← i
        while j > 0 and A[j-1] > x
                // As we scan for insertion point, we move elements
                // to the right.
                A[j] ← A[j-1]
                j ← j - 1
        end while

        // Finally, write the element being inserted into its final spot.
        A[j] ← x
end for
```



Insertion Sort

Let's work through an Insertion Sort. We'll start with this unsorted array.

9	13	6	7	1	16	3	12

(We're going to work through the "swapless edition" where we save the element under consideration in a variable x, and copy elements to the right, as opposed to the "swapping" edition which swaps the element to insert repeatedly to the left.)

Insertion Sort

i = 1 (the **second** element)

Current value to insert is 13. Scan left for insertion point, i.e. j such that A[j] > A[i]

9	13	6	7	1	16	3	12

As we scan, we move elements one slot to the right. The scan ends when insertion point is found.

9	13	6	7	1	16	3	12

Since 13 > 9, the scan stops immediately and no change is made. The insertion point was 1.

Insertion Sort

i = 2

Scan left for insertion point for 6, moving elements one to the right as we go.

9	13	6	7	1	16	3	12

13 > 6 and 9 > 6, so 13 and 9 move one slot to the right, and 6 is inserted at slot 0.

6	9	13	7	1	16	3	12

Insertion Sort

i = 3

Scan left for insertion point for 7. 13 > 7 and 9 > 7, but 6 < 7, so scan stops at index 1.

6	9	13	7	1	16	3	12

As the scan proceeded, it moved 13 and 9 to the right, making room for 7.

6	7	9	13	1	16	3	12

Insertion Sort

i = 4

Find insertion point for 1. 13 > 1, 9 > 1, 7 > 1, 6 > 1, so we scan all the way to index 0.

6	7	9	13	1	16	3	12

The 1 is written to slot 0, and 6, 7, 9, and 13 have been moved to the right.

1	6	7	9	13	16	3	12

Insertion Sort

i = 5

Find where 16 should be inserted. 16 > 13, so the scan stops immediately.

1	6	7	9	13	16	3	12

When the scan stops immediately, it means the insertion point is where the element already is.

1	6	7	9	13	16	3	12

Insertion Sort

i = 6

Find insertion point for 3. We scan past 16, 13, 9, 7, and 6, but stop at 1 because 1 < 3.

1	6	7	9	13	16	3	12

During the scan, 16, 13, 9, 7, and 6 moved to the right, making room for 3.

1	3	6	7	9	13	16	12

Insertion Sort

i = 7

Find insertion point for 12. 16 > 12 and 13 > 12, so we move 16 and 13 to the right.

1	3	6	7	9	13	16	12

12 > 9, so the scan stops and the 12 is written at slot 5.

1	3	6	7	9	12	13	16

Aaaaand... Done.

Insertion Sort

- What input array would result in the **best case** running time?
- What input array would result in the **worst case** running time?
- Compare to Selection Sort's best and worst case running time.

Insertion Sort: Average Case

- The average case takes some math to calculate, but works out to $O(N^2)$.
- However, it works out to half the comparisons, on average, that Selection Sort does.
- Selection Sort **always** does the same number of comparisons, but varies in number of swaps.
- Insertion Sort can do as many comparisons as Selection Sort, but does about half the comparisons on a random input array.
- So, not all $O(N^2)$ are alike... the running time is **bounded** by N^2, but different $O(N^2)$ algorithms behave differently, sometimes much better or worse.

Insertion Sort: Is it good? Do people actually use it?

- Insertion Sort is faster in practice than other quadratic algorithms such as Selection Sort or Bubble Sort.
- Insertion Sort is actually one of the fastest known algorithms for sorting **very small** arrays.
  (Around <=10 items.)
- Quicksort is the common "fast" algorithm used in many standard libraries for any size input, but sometimes the sort method will switch to Insertion Sort for very small inputs.

Exercise: Build your own Insertion Sort

- Repl.it: InsertionSort
- This is similar to the SelectionSort exercise. We'll use the Fortune 500 data set again.
- You will be sorting not just by company name, but by all of the other fields! Look at the getField() method in Record.java.
- You will be sorting in **ascending** and **descending** order.
- Your Insertion Sort shouldn't use any ArrayList magic like add(obj, index)... so the code uses plain arrays, not ArrayList, to be sure.
- Last week, we had to pay attention to **case sensitivity**. This time, we need to do that as well, but we also need to flip the comparison when sorting in descending order. How would you "reverse" the output of String.compareTo?

```java
private int compareRecord(Record record1, Record record2, String fieldName, boolean ascending) {
  String value1 = record1.getField(fieldName);
  String value2 = record2.getField(fieldName);
  int compareResult = value1.compareToIgnoreCase(value2);
  if (!ascending) {
    compareResult = -compareResult;
  }
  return compareResult;
}
```

```java
private void insertionSort(Record[] records,
                           String fieldName,
                           boolean ascending) {
  for (int i=1, n=records.length; i<n; i++) {
    Record temp = records[i];
    int j = i;
    while (j > 0 && compareRecord(records[j-1], temp, fieldName, ascending) > 0) {
      records[j] = records[j-1];
      j--;
    }
    records[j] = temp;
  }
}
```