2023-01-27

Unit 7 Review

7.1: `ArrayList`

ArrayList

- `ArrayLists` are collections of values of the same Object type; But have different declaration syntax than Arrays; **Primitive types (int, boolean, double, etc.) are not supported**

```
ArrayList<type> name;
```

- Examples

```
ArrayList<boolean> answers; ** PRIMITIVE TYPES UNSUPPORTED **
ArrayList<Boolean> answers;
ArrayList<int> scores; ** PRIMITIVE TYPES UNSUPPORTED **
ArrayList<Integer> scores;
ArrayList<String> questions;
ArrayList<Student> students;
```

- **Important:** You must import `ArrayList` prior to using it

```
import java.util.ArrayList;
```

Generics / Generic Types

- `ArrayList` is an example of a class that uses a Generic Type

```
ArrayList<type> name;
```

- Generic Types are an option when the **same code** can be used across a variety of data types – and frees you from needing to create an overloaded method for every type
- `ArrayList` is able to use Generic Types because the internals assume everything is a `Object` type (and all `Object` types share the functionality required for `ArrayList` to work)
- You can read more about Generics in the online Java documentation
  - [Oracle Java Documentation: Why Use Generics?]

ArrayList

- Like Arrays, you must initialize `ArrayLists` prior to using them; The most common usage is with the no-parameter Constructor

```
ArrayList<Boolean> answers  = new ArrayList<Boolean>();
ArrayList<Integer> scores = new ArrayList<Integer>();
ArrayList<String> questions = new ArrayList<String>();
ArrayList<Student> students = new ArrayList<Student>();
```

- **Note:** There are two other `ArrayList` Constructors that you can explore on your own

```
ArrayList<type> name = new ArrayList<type>(Collection<type> c);
ArrayList<type> name = new ArrayList<type>(init initialCapacity);
```

`ArrayList`

- Unlike Arrays, `ArrayLists` automatically manage their memory usage as you `ArrayList.add()` and `ArrayList.remove()` elements to/from the the `ArrayList`
- Unlike Arrays, `ArrayLists` do not have a `length` property that indicates the fixed-size of the Array; They have the `ArrayList.size()` method that indicates the current number of elements included in the `ArrayList`
- `ArrayLists` have an internal capacity - which you cannot access - that grows and shrinks as needed to ensure elements can be quickly added. **The default capacity is 10.**
- The capacity is adjusted to ensure that the there is enough free space to quickly accommodate new items via `ArrayList.add()`; But not so much excess free space that available memory is wasted

Array vs `ArrayList`

Array

true	false	true	*false*	*false*	*false*	*false*	*false*	*false*	*false*
0	1	2	3	4	5	6	7	8	9

```
boolean[] answers = new boolean[10];
answers[0] = true; answers[1] = false; answers[2] = true;
answers.length == 10
answers[3-9] are set to default values
```

ArrayList

true	false	true							
0	1	2	3	4	5	6	7	8	9

```
ArrayList<Boolean> answers = new ArrayList<Boolean>();
answers.add(true); answers.add(false); answers.add(true);
answers.size() == 3
answers[3-9] are unused pre-allocated capacity
```

`ArrayList` Methods

`ArrayList` Methods

- **`add()`**
- `clear()`
- `get()`
- `isEmpty()`
- `remove()`
- `removeRange()`
- `set()`
- `size()`

`ArrayList` `index` values are zero-based (just like Arrays)

Signatures

- `boolean add(E obj)`
- `void add(int index, E obj)`

Overview

- Add an item either to the end of the `ArrayList` (**always returns `true`**) or at the specified `index` (existing items will shift right; their index values will increase by 1)
  - The first version of **`add()`** always returns `true` because `ArrayList` implements the `Collection` interface - which **can** be implemented by other classes to restrict the creation of duplicate or `null` elements (`ArrayList` has no such restrictions)
- Automatically increases the `ArrayList` capacity as needed
- Will throw IndexOutOfBoundsException if `index` is out of range `(index < 0 || index > size())`

`ArrayList` Methods

- `add()`
- **`clear()`**
- `get()`
- `isEmpty()`
- `remove()`
- `removeRange()`
- `set()`
- `size()`

Signatures

- **`void clear`**`()`

Overview

- Removes all elements from the `ArrayList`
- After this call `ArrayList.size() == 0`
- Automatically decreases the `ArrayList` capacity as needed

`ArrayList` Methods

- `add()`
- `clear()`
- **`get()`**
- `isEmpty()`
- `remove()`
- `removeRange()`
- `set()`
- `size()`

`ArrayList index` values
are zero-based
(just like Arrays)

Signatures

- **`E get`**`(`**`int`**` index)`

Overview

- Returns the element at the specified position in the `ArrayList`
- You must use this method to access the items in an `ArrayList`; `ArrayList` does not support the `[]` syntax of Arrays
- Will throw IndexOutOfBoundsException if `index` is out of range `(index < 0 || index >= size())`

`ArrayList` Methods

- `add()`
- `clear()`
- `get()`
- **`isEmpty()`**
- `remove()`
- `removeRange()`
- `set()`
- `size()`

Signatures

- **`boolean isEmpty`**`()`

Overview

- Returns true if the `ArrayList` has no items

`ArrayList` Methods

- `add()`
- `clear()`
- `get()`
- `isEmpty()`
- **`remove()`**
- `removeRange()`
- `set()`
- `size()`

`ArrayList` `index` values are zero-based (just like Arrays)

Signatures

- `boolean remove(Object obj)`
- `E remove(int index)`

Overview

- Removes the first item from the `ArrayList` that matches `obj`; or at the specified `index` (existing items will shift left; their index values will decrease by 1)
    - `remove(Object obj)` returns `true/false` if an element in the `ArrayList` returns `true` for `obj.equals(element)` `(or obj == null == element)` and was removed
        - Note: Does **not use** Object equality (`obj == element`)
    - `remove(int index)` returns the element that was removed from the `ArrayList`
- Automatically decreases the `ArrayList` capacity as needed
- Will throw IndexOutOfBoundsException if `index` is out of range `(index < 0 || index >= size())`

`ArrayList` Methods

- `add()`
- `clear()`
- `get()`
- `isEmpty()`
- `remove()`
- **`removeRange()`**
- `set()`
- `size()`

`ArrayList index` values are zero-based (just like Arrays)

Signatures

- `void removeRange(int fromIndex, int toIndex)`

Overview

- Removes all of the elements whose index is between `fromIndex` (inclusive) and `toIndex` (exclusive). Shifts any succeeding elements to the left (reduces their index).
- Automatically decreases the `ArrayList` capacity as needed
- Will throw IndexOutOfBoundsException if `fromIndex` or `toIndex` is out of range `(fromIndex < 0 || fromIndex >= size() || toIndex > size() || toIndex < fromIndex)`

`ArrayList` Methods

- `add()`
- `clear()`
- `get()`
- `isEmpty()`
- `remove()`
- `removeRange()`
- **`set()`**
- `size()`

`ArrayList` `index` values are zero-based (just like Arrays)

Signatures

- `E set(int index, E element)`

Overview

- Replaces the element at the specified position in this `ArrayList` with the specified element.
- Returns the element that was removed from the `ArrayList` at `index`
- You must use this method to access the items in an `ArrayList`; `ArrayList` does not support the `[]` syntax of Arrays
- Will throw IndexOutOfBoundsException if `index` is out of range `(index < 0 || index >= size())`

`ArrayList` Methods

- `add()`
- `clear()`
- `get()`
- `isEmpty()`
- `remove()`
- `removeRange()`
- `set()`
- **`size()`**

Signatures

- **`int size`**`()`

Overview

- Returns the number of elements in this `ArrayList`

7.3: Traversing `ArrayLists` with Loops

Traversing `ArrayLists`

- `ArrayLists` support the same mechanisms you used when traversing Arrays - `while`, `for`, `for-each` - with the following differences

Operation	**`Array`**	**`ArrayList`**
length/size	`Array.length` (property)	`ArrayList.size()` (method)
read	`value = array[index];`	`value = arrayList.get(index);`
write	`array[index] = value;`	`arrayList.set(index, value);`

Traversing `ArrayLists`

Array - for loop

<pre>Integer[] array = {1, 2, 3, 4, 5};
for (int idx = 0 ; idx < array.length ; idx++) {
  int value = array[idx];
  System.out.println(value);
  array[idx] = value + 1;
}</pre>

ArrayList - for loop

<pre>Integer[] array = {1, 2, 3, 4, 5};
ArrayList<Integer> arrayList = new ArrayList<Integer>(Arrays.asList(array));
for (int idx = 0 ; idx < arrayList.size() ; idx++) {
  int value = arrayList.get(idx);
  System.out.println(value);
  arrayList.set(idx, value + 1);
}</pre>

Traversing `ArrayLists`

Array & ArrayList - for loop **ArrayIndexOutOfBoundsException**
This exception will be thrown if you try to access the item at an index less than 0 or greater than the number of items in the Array or ArrayList

Traversing `ArrayLists`

Array - for-each loop

`Integer[] array = {1, 2, 3, 4, 5};`
`for (Integer value : array) {`
  `System.out.println(value);`
`}`

ArrayList - for-each loop

`Integer[] array = {1, 2, 3, 4, 5};`
`ArrayList<Integer> arrayList = new ArrayList<Integer>(Arrays.asList(array));`
`for (Integer value : arrayList) {`
  `System.out.println(value);`
`}`

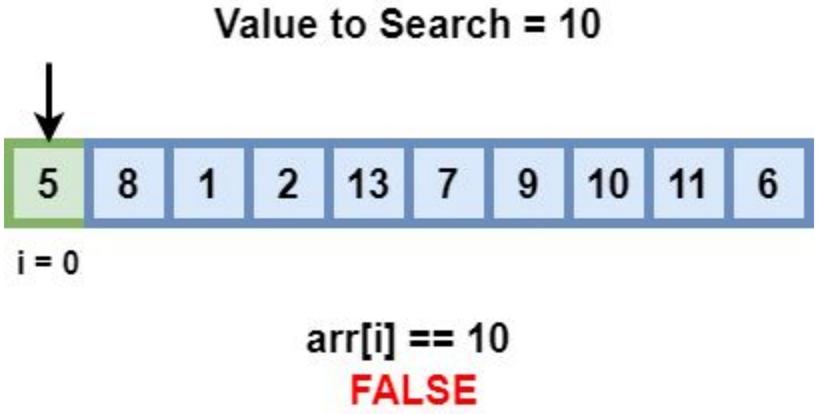
Traversing `ArrayLists`

ArrayList for-each loop **ConcurrentModificationException**
This exception will be thrown if you try to add or remove items from an ArrayList while traversing that ArrayList with a for-each loop

7.5
Search Algorithms

Sequential Search (aka Linear Search)

```
for i ← 0 to length(array)-1
        if array[i] == targetValue
                return i
        end if
end for
return not found
```

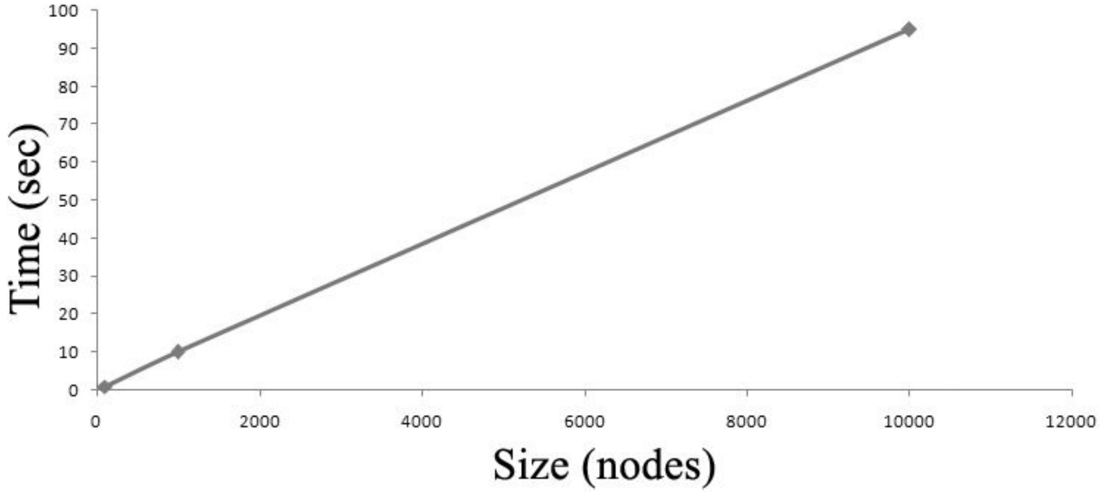


Pro: You can search for an item in any unsorted list and it is guaranteed to be found if it exists in the array

Con: If you're searching a long list, this can be a very slow approach

Sequential Search

If your list contains N elements, and the target element is at a random position in the list then on average it will take N/2 checks to find your element. When the time an algorithm takes to complete is directly related to the size of the array (N), we call this algorithm **linear** and the notation is O(N).



Binary Search: Divide and Conquer

Binary search is a much faster algorithm, but requires that the list be **sorted**.

We can use the fact that the list is sorted to reduce the problem to smaller problems.

An algorithm that breaks the problem into smaller sub-problems is called a **divide-and-conquer algorithm**.

1	3	6	7	9	12	13	16

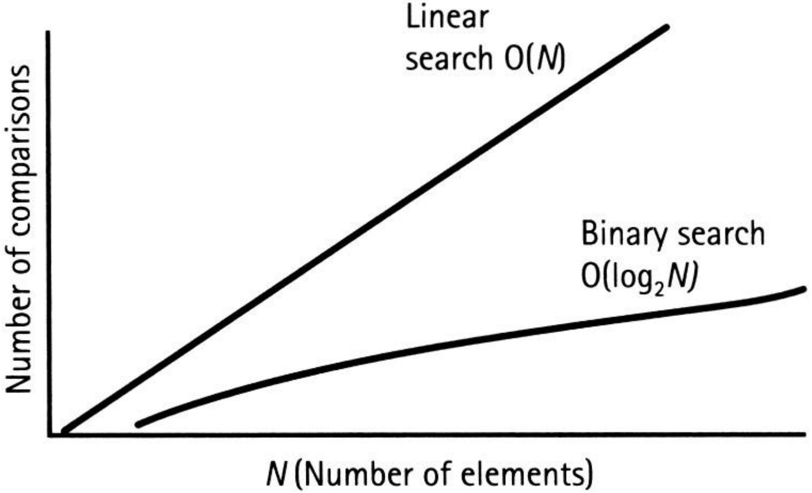
Binary Search Algorithm

```
// Precondition: array must be in sorted order
low ← 0, high ← length(array)-1
while low <= high
	middle ← (low + high) / 2
	if array[middle] == targetValue
		return middle
	else if array[middle] < targetValue
		low ← middle + 1
	else
		high ← middle - 1
	end if
end while
return not found
```

Let's run through it on the whiteboard...

Binary Search

If your list contains N elements, and the target element is at a random position in the list then on average it will take $\log_2 N$ checks to find your element. To give you a sense of this speedup, the $\log_2 1024 = 10$ (since $2^{10} = 1024$). **This means that binary search runs 10x faster on sorted lists of length ~1K.** The notation for this algorithm is O(log N)



Comparing Sequential Search vs Binary Search (These are "worst case" numbers.)

N	**Sequential Search Comparison Count**	**Binary Search Comparison Count**
2	2	2
4	4	3
8	8	4
16	16	5
100	100	7

7.6.1
Selection Sort

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;
}
```

Selection Sort

- Selection Sort is an algorithm that relies on **swapping** array elements.
- We used swapping in section 6.4 to reverse the elements in an array.
- Java doesn't have a built-in "swap" capability, so you must use a temporary variable:

```
// Swap array elements array[i] and array[j]
int temp = array[i];
array[i] = array[j];
array[j] = temp;
```

Swapping variables is also called exchanging variables. Sort algorithms that rely on swapping/exchanging values are classified as **exchange sorts**. (Confusingly, there is also a specific sort algorithm called Exchange Sort.)

Selection Sort Algorithm

```
for i ← 0 to length(array)-1
    // Loop invariant: The array to the left of i contains the i smallest values
    // in the array, in sorted order.
    jMin ← i
    for j ← i+1 to length(array)-1
        // Loop invariant: A[jMin] is smallest value in range A[i]...A[j-1]
        if A[j] < A[jMin] then jMin ← j
    end for
    if jMin != i then swap A[i], A[jMin]
end for
```

Let's run through it on the whiteboard...

Selection Sort – an intuitive but slow algorithm

How many comparisons does Selection Sort do?

It does N-1 comparisons on the first pass,
N-2 comparisons on the second pass,
N-3 comparisons on the third pass, and so on.

$$1 + 2 + 3 + \ldots + (N-1) = \sum_{i=1}^{N-1} i$$

Loop #	Comparisons (N=8)
1	7
2	6
3	5
4	4
5	3
6	2
7	1

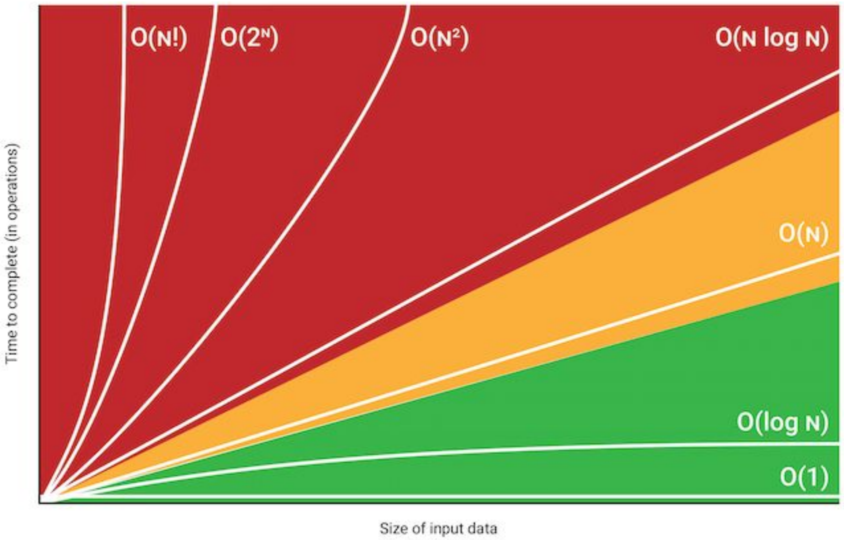
Selection Sort – our old friend O(N^2)

$$1 + 2 + 3 + \ldots + (N-1) = \sum_{i=1}^{N-1} i$$

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

$$\sum_{i=1}^{N-1} i = \frac{(N-1)\cdot N}{2}$$

$$\frac{(N-1)\cdot N}{2} = \frac{N^2 - N}{2} \approx O(N^2)$$



Selection Sort – an intuitive but slow algorithm

$O(N^2)$ is also known as **quadratic time**. Many simple sort algorithms are quadratic time.

It can be OK for a small number of elements, but as N gets big, the algorithm's running time becomes very long.

In Unit 10, we will learn a sorting algorithm, **merge sort**, that has much better running time: $O(N \log N)$. But it's also more complicated to code!

N	N^2
1	1
10	100
100	10,000
1000	1,000,000
10000	100,000,000
100000	10,000,000,000
1000000	10^{12}

Selection Sort

Best case: Does $(N^2-N)/2$ comparisons, 0 swaps

Worst case: Does $(N^2-N)/2$ comparisons, N swaps

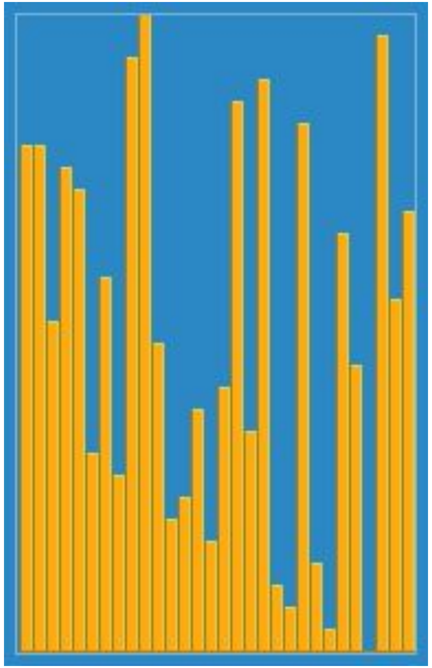
Average case: Does $(N^2-N)/2$ comparisons, N/2 swaps

7.6.2
Insertion Sort

Introducing Insertion Sort

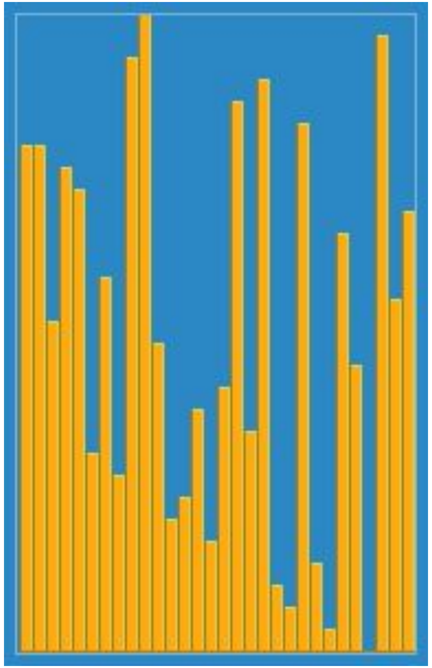
Insertion Sort is more complex than Selection Sort, but is much faster when the data is partially sorted.

- Insertion Sort is still a quadratic algorithm, that is, $O(N^2)$.
- Insertion Sort is faster in practice than other quadratic algorithms such as Selection Sort.
- Insertion Sort is actually one of the fastest known algorithms for sorting **very small** arrays. (Around <=10 items.)



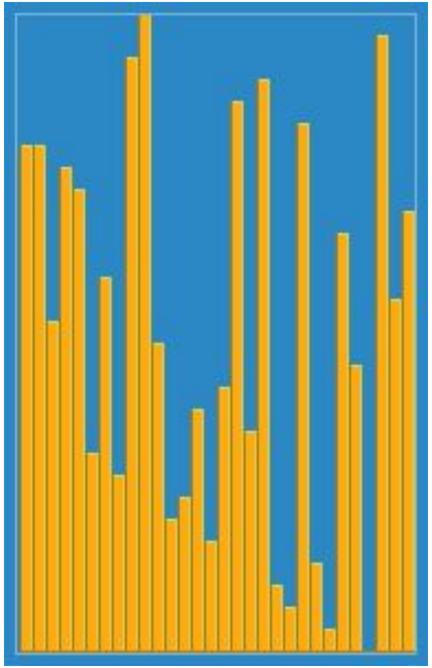
Insertion Sort Algorithm

- 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)-1
    // 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
```



Let's run through it on the whiteboard...

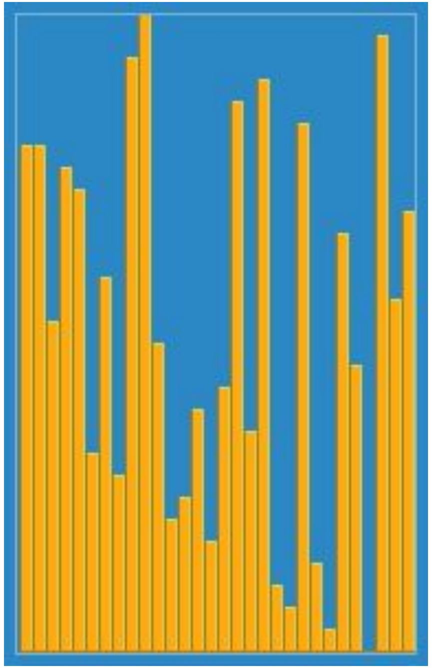
Insertion Sort Algorithm: Swapless Edition

```
// A bit more efficient than swapping. Less read/write operations.
// Note we start with second element i = 1
for i ← 1 to length(array)-1
        // 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

Best case: an already sorted array

(Does well on partially sorted arrays too)

Worst case: an entirely reversed array

Average case: about half the comparisons of Selection Sort