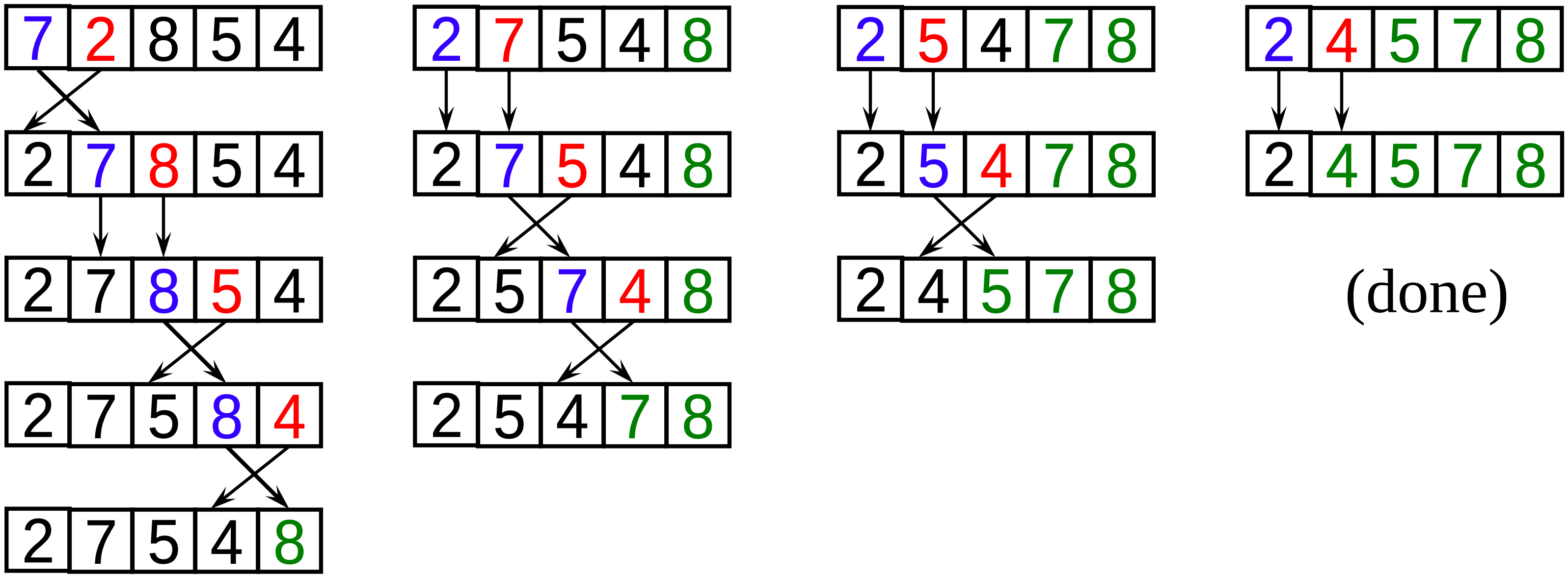
Simple Sorting Algorithms

Bubble sort

- Compare each element (except the last one) with its neighbor to the right
  - If they are out of order, swap them
  - This puts the largest element at the very end
  - The last element is now in the correct and final place
- Compare each element (except the last *two*) with its neighbor to the right
  - If they are out of order, swap them
  - This puts the second largest element next to last
  - The last two elements are now in their correct and final places
- Compare each element (except the last *three*) with its neighbor to the right
  - Continue as above until you have no unsorted elements on the left

Example of bubble sort



Code for bubble sort

- ```
  public static void bubbleSort(int[] a) {
     int outer, inner;
     for (outer = a.length - 1; outer > 0; outer--) {  // counting down
        for (inner = 0; inner < outer; inner++) {        // bubbling up
           if (a[inner] > a[inner + 1]) {  // if out of order...
              int temp = a[inner];          // ...then swap
              a[inner] = a[inner + 1];
              a[inner + 1] = temp;
           }
        }
     }
  }
  ```
  ```

Analysis of bubble sort

- ```
  for (outer = a.length - 1; outer > 0; outer--) {
    for (inner = 0; inner < outer; inner++) {
      if (a[inner] > a[inner + 1]) {
        // code for swap omitted
  } } }
  ```
- Let n = a.length = size of the array
- The outer loop is executed n-1 times (call it n, that's close enough)
- Each time the outer loop is executed, the inner loop is executed
  - Inner loop executes n-1 times at first, linearly dropping to just once
  - On average, inner loop executes about n/2 times for each execution of the outer loop
  - In the inner loop, the comparison is always done (constant time), the swap might be done (also constant time)
- Result is n * n/2 + k, that is, O(n2/2 + k) = $O(n^2)$
  ```

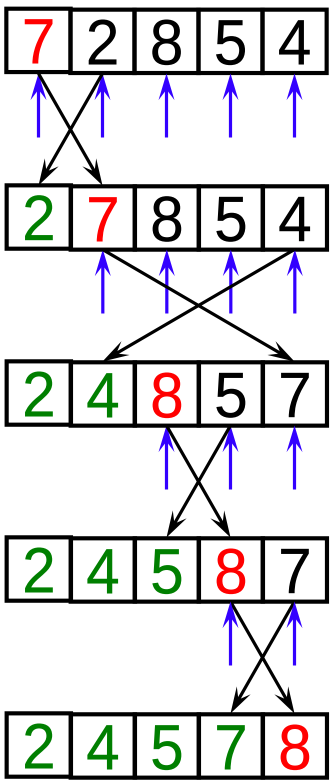
Loop invariants

- You run a loop in order to change things
- Oddly enough, what is usually most important in understanding a loop is finding an invariant: that is, *a condition that doesn't change*
- In bubble sort, we put the largest elements at the end, and once we put them there, we don't move them again
  - The variable `outer` starts at the last index in the array and decreases to `0`
  - Our invariant is: Every element to the right of `outer` is in the correct place
  - That is, `for all j > outer, if i < j, then a[i] <= a[j]`
  - When this is combined with the loop exit test, `outer == 0`, we know that *all* elements of the array are in the correct place

Selection sort

- Given an array of length n,
  - Search elements 0 through n-1 and select the smallest
    - Swap it with the element in location 0
  - Search elements 1 through n-1 and select the smallest
    - Swap it with the element in location 1
  - Search elements 2 through n-1 and select the smallest
    - Swap it with the element in location 2
  - Search elements 3 through n-1 and select the smallest
    - Swap it with the element in location 3
  - Continue in this fashion until there's nothing left to search

Example and analysis of selection sort



- The selection sort might swap an array element with itself--this is harmless, and not worth checking for
- Analysis:
  - The outer loop executes n-1 times
  - The inner loop executes about n/2 times on average (from n to 2 times)
  - Work done in the inner loop is constant (swap two array elements)
  - Time required is roughly (n-1)*(n/2)
  - You should recognize this as $O(n^2)$

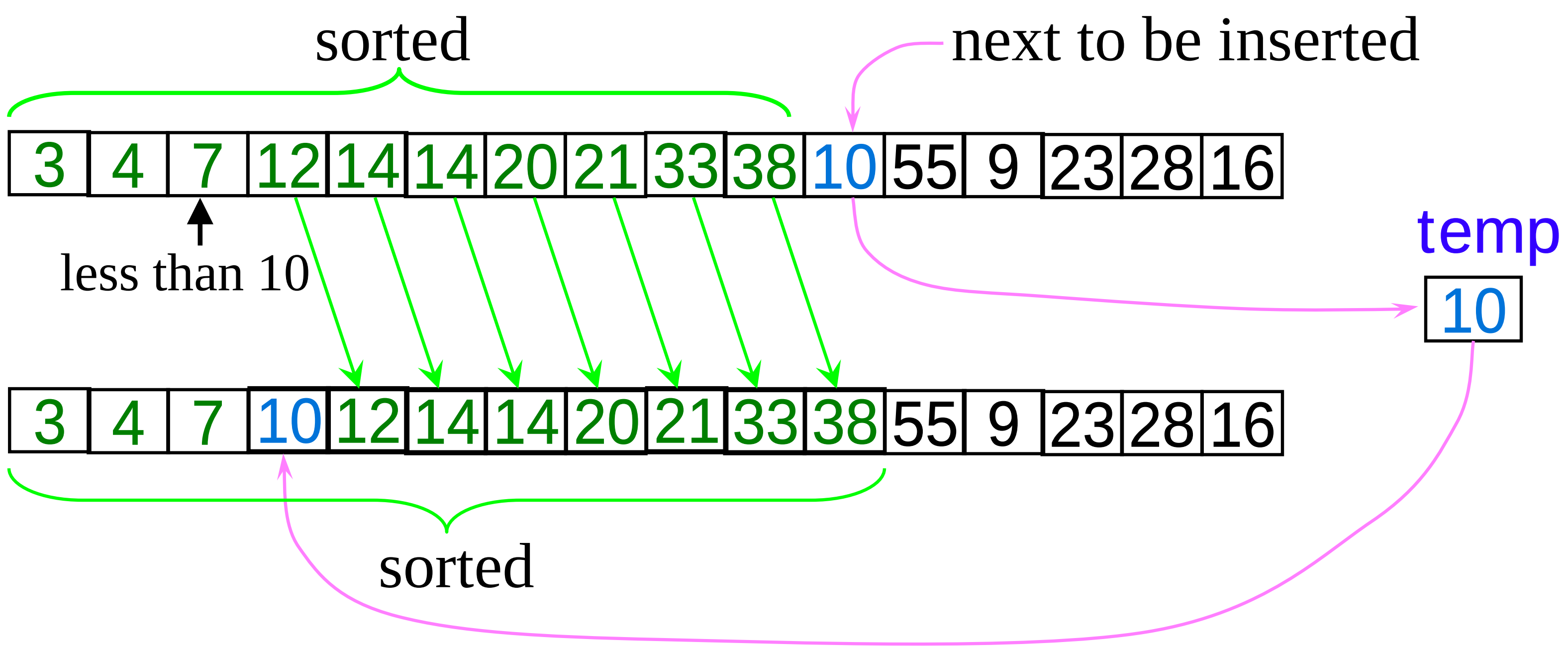
Code for selection sort

```
public static void selectionSort(int[] a) {
        int outer, inner, min;
        for (outer = 0; outer < a.length - 1; outer++) {
            min = outer;
            for (inner = outer + 1; inner < a.length; inner++) {
                if (a[inner] < a[min]) {
                    min = inner;
                }
            }
            // a[min] is least among a[outer]..a[a.length - 1]
            int temp = a[outer];
            a[outer] = a[min];
            a[min] = temp;
        }
    }
```

Insertion sort

- The outer loop of insertion sort is:
  `for (outer = 1; outer < a.length; outer++) {...}`
- The invariant is that all the elements to the left of `outer` are sorted with respect to one another
  - For all i < outer, j < outer, if i < j then a[i] <= a[j]
  - This does *not* mean they are all in their final correct place; the remaining array elements may need to be inserted
  - When we increase `outer`, `a[outer-1]` becomes to its left; we must keep the invariant true by inserting `a[outer-1]` into its proper place
  - This means:
    - Finding the element's proper place
    - Making room for the inserted element (by shifting over other elements)
    - Inserting the element

One step of insertion sort



Analysis of insertion sort

- We run once through the outer loop, inserting each of n elements; this is a factor of n
- On average, there are $n/2$ elements already sorted
  - The inner loop looks at (and moves) half of these
  - This gives a second factor of $n/4$
- Hence, the time required for an insertion sort of an array of n elements is proportional to $n^2/4$
- Discarding constants, we find that insertion sort is $O(n^2)$

Summary

- Bubble sort, selection sort, and insertion sort are all $O(n^2)$
- As we will see later, we can do much better than this with somewhat more complicated sorting algorithms
- Within $O(n^2)$,
  - Bubble sort is very slow, and should probably never be used for anything
  - Selection sort is intermediate in speed
  - Insertion sort is usually the fastest of the three--in fact, for small arrays (say, 10 or 15 elements), insertion sort is faster than more complicated sorting algorithms
- Selection sort and insertion sort are “good enough” for small arrays