Sorting Algorithms

Part 2

Overview

- Review of sorting & desirable properties for sorting algorithms
- Introduction to simple sorting algorithms
  - Bubble Sort
  - Selection Sort
  - Insertion Sort

Review of sorting

- Sorting – arrange a collection of items according to some pre-defined ordering rules

- Desirable properties for sorting algorithms
  - Stability – preserve order of already sorted input
  - Good run time efficiency (in the best, average or worst case)
  - In-place sorting – if memory is a concern
  - Suitability – the properties of the sorting algorithm are well-matched to the class of input instances which are expected i.e. consider specific strengths and weaknesses when choosing a sorting algorithm

Overview of sorting algorithms

Algorithm	Best case	Worst case	Average case	Space Complexity	Stable?
Bubble Sort	n	n^2	n^2	1	Yes
Selection Sort	n^2	n^2	n^2	1	No
Insertion Sort	n	n^2	n^2	1	Yes
Merge Sort	$n \log n$	$n \log n$	$n \log n$	$O(n)$	Yes
Quicksort	$n \log n$	n^2	$n \log n$	n (worst case)	No*
Heapsort	$n \log n$	$n \log n$	$n \log n$	1	No
Counting Sort	$n + k$	$n + k$	$n + k$	$n + k$	Yes
Bucket Sort	$n + k$	n^2	$n + k$	$n \times k$	Yes
Timsort	n	$n \log n$	$n \log n$	n	Yes
Introsort	$n \log n$	$n \log n$	$n \log n$	$\log n$	No

*the standard Quicksort algorithm is unstable, although stable variations do exist

Comparison sorts

- A comparison sort is a type of sorting algorithm which uses comparison operations only to determine which of two elements should appear first in a sorted list.
- A sorting algorithm is called ***comparison-based*** if the only way to gain information about the total order is by comparing a pair of elements at a time via the order $\leq$.
- The simple sorting algorithms which we will discuss in this lecture (Bubble Sort, Insertion Sort, and Selection Sort) all fall into this category.
- A fundamental result in algorithm analysis is that no algorithm that sorts by comparing elements can do better than $n \log n$ performance in the average or worst cases.
- Non-comparison sorting algorithms (e.g. Bucket Sort, Counting Sort, Radix Sort) can have better worst-case times.

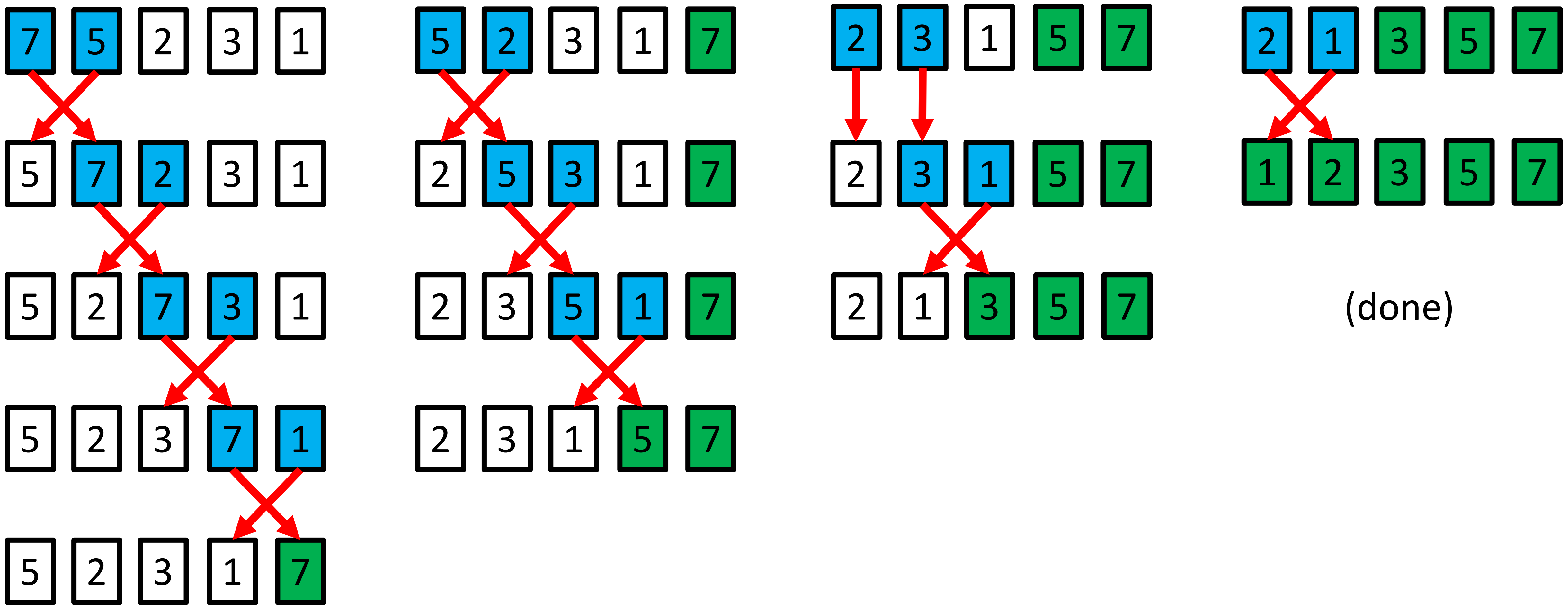
Bubble Sort

- Named for the way larger values in a list “bubble up” to the end as sorting takes place
- Bubble Sort was first analysed as early as 1956 (time complexity is n in best case, and n^2 in worst and average cases)
- Comparison-based
- In-place sorting algorithm (i.e. uses a constant amount of additional working space in addition to the memory required for the input)
- Simple to understand and implement, but it is slow and impractical for most problems even when compared to Insertion Sort
- Can be practical in some cases on data which is nearly sorted

Bubble Sort procedure

- Compare each element (except the last one) with its neighbour 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 neighbour 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 neighbour to the right
  - …
- Continue as above until there are no unsorted elements on the left

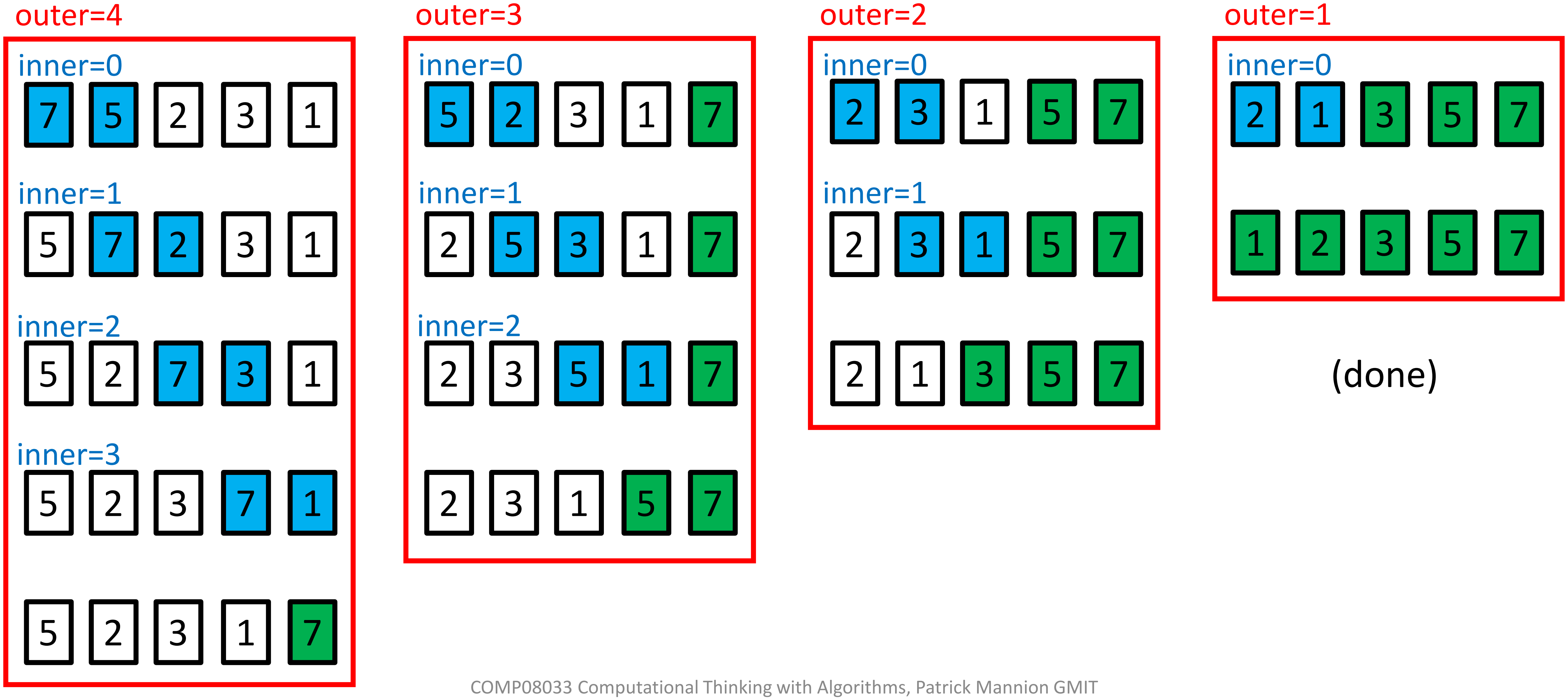
Bubble Sort example



Bubble Sort in Java

```java
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;
      }
    }
  }
}
```

Bubble Sort example



Analysing Bubble Sort (worst case)

```
for (outer = a.length - 1; outer > 0; outer--) {
  for (inner = 0; inner < outer; inner++) {
    if (a[inner] > a[inner + 1]) {
      //swap code omitted
    }
  }
}
```

- In the worst case, the outer loop executes n-1 times (say n times)
- On average, inner loop executes about n/2 times for each outer loop
- In the inner loop, comparison and swap operations take constant time k
- Result is $n \times \frac{n}{2} + k = \frac{n^2}{2} + k \approx O(n^2)$

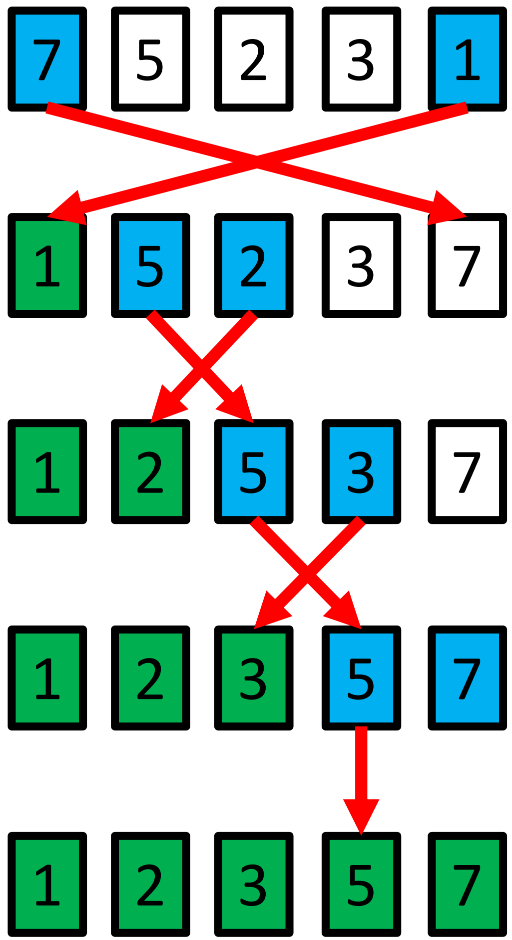
Selection Sort

- Comparison-based
- In-place
- Unstable
- Simple to implement
- Time complexity is n^2 in best, worst and average cases
- Generally gives better performance than Bubble Sort, but still impractical for real world tasks with a significant input size
- In every iteration of Selection Sort, the minimum element (when using ascending order) from the unsorted subarray on the right is picked and moved to the sorted subarray on the left

Selection Sort procedure

- 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

Selection Sort example



The element at index 4 is the smallest, so swap with index 0

The element at index 2 is the smallest, so swap with index 1

The element at index 3 is the smallest, so swap with index 2

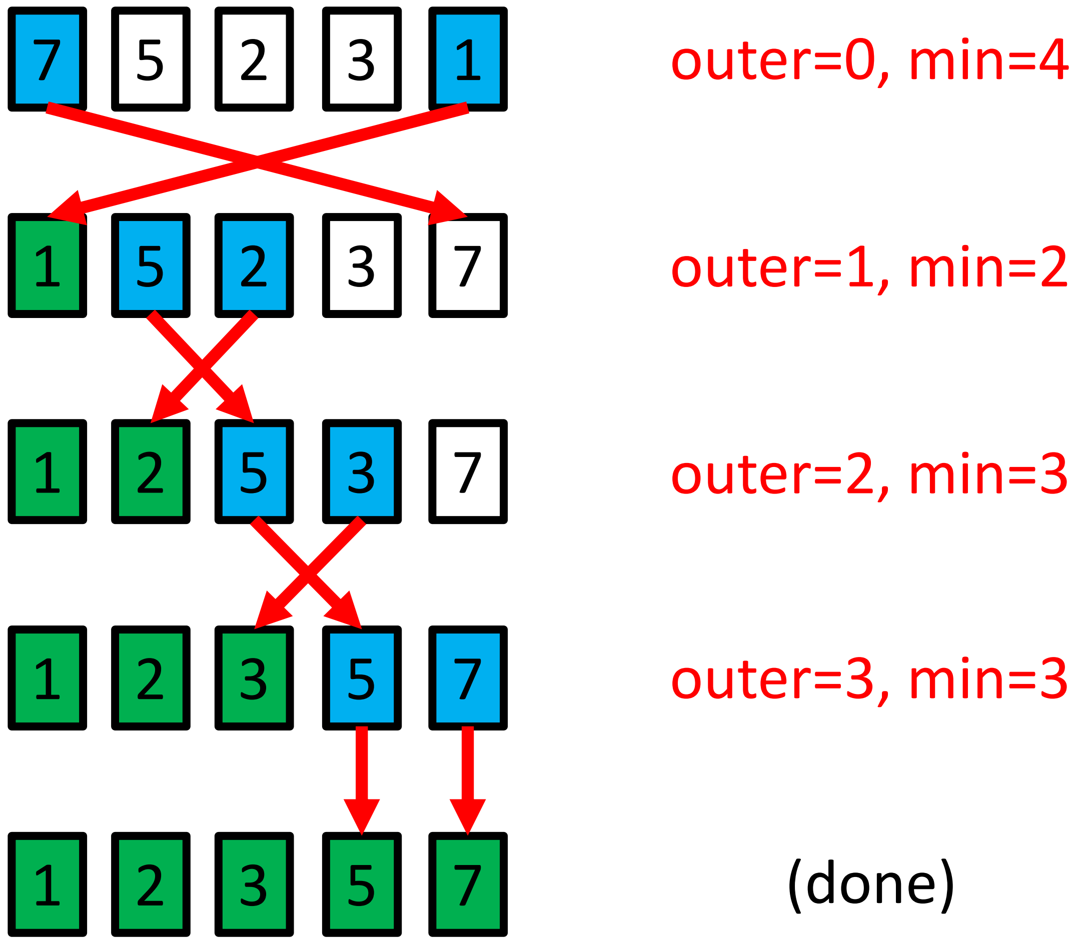
The element at index 3 is the smallest, so swap with index 3. Selection Sort might swap an array element with itself; this is harmless, and not worth checking for

Selection Sort in Java

```
public static void selectionSort(int[] a) {
  int outer=0, inner=0, min=0;
  for (outer = 0; outer < a.length - 1; outer++) { // outer counts up
    min = outer;
    for (inner = outer + 1; inner < a.length; inner++) {
      if (a[inner] < a[min]) { // find index of smallest value
        min = inner;
      }
    }

    // swap a[min] with a[outer]
    int temp = a[outer];
    a[outer] = a[min];
    a[min] = temp;
  }
}
```

Analysing Selection Sort



- The outer loop runs $n - 1$ times
- The inner loop executes about n/2 times on average (from n to 2 times)
- Result is $(n - 1) \times \frac{n}{2} \approx n^2$ in best, worst and average cases

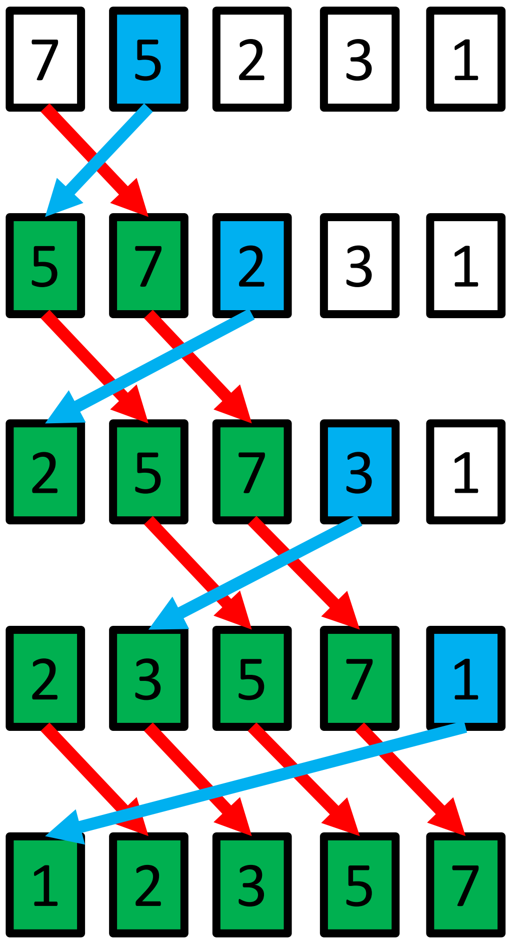
Insertion Sort

- Similar to the method usually used by card players to sort cards in their hand.
- Insertion Sort is easy to implement, stable, in-place, and works well on small lists and lists that are close to sorted.
- On data sets which are already substantially sorted it runs in $n + d$ time, where d is the number of inversions.
- However, it is very inefficient for large random lists.
- Insertion Sort is iterative and works by splitting a list of size n into a head ("sorted") and tail ("unsorted") sublist.

Insertion Sort procedure

- Start from the left of the array, and set the “key” as the element at index 1. Move any elements to the left which are > the “key” right by one position, and insert the “key”.
- Set the “key” as the element at index 2. Move any elements to the left which are > the key right by one position and insert the key.
- Set the “key” as the element at index 3. Move any elements to the left which are > the key right by one position and insert the key.
- …
- Set the “key” as the element at index n-1. Move any elements to the left which are > the key right by one position and insert the key.
- The array is now sorted

Insertion Sort example



a[1]=5 is the key; 7>5 so move 7 right by one position, and insert 5 at index 0

a[2]=2 is the key; 7>2 and 5>2 so move both 7 and 5 right by one position, and insert 2 at index 0

a[3]=3 is the key; 7>3 and 5>3 so move both 7 and 5 right by one position, and insert 3 at index 1

a[4]=1 is the key; 7>1, 5>1, 3>1 and 2>1 so move 7, 5, 3 and 2 right by one position, and insert 1 at index 0

(done)

Insertion Sort in Java

```java
public static void insertionSort(int a[]) {
  for (int i=1; i<a.length; i++) {
    int key = a[i]; // value to be inserted
    int j = i-1;
    //move all elements > key right one position
    while (j>=0 && a[j] > key) {
      a[j+1] = a[j];
      j = j-1;
    }
    a[j+1] = key; //insert key in its new position
  }
}
```

Analysing Insertion Sort

- The total number of data comparisons made by Insertion Sort is the number of inversions d plus at most $n - 1$
- A sorted list has no inversions – therefore Insertion Sort runs in linear $\Omega(n)$ time in the best case (when the input is already sorted)
- On average, a list of size n has $\frac{(n-1)\times n}{4}$ inversions, and the number of comparisons is $n - 1 + \frac{(n-1)\times n}{4} \approx n^2$
- In the worst case, a list of size n has $\frac{(n-1)\times n}{2}$ inversions (reverse sorted input), and the number of comparisons is $n - 1 + \frac{(n-1)\times n}{2} \approx O(n^2)$

Comparison of simple sorting algorithms

- The main advantage that Insertion Sort has over Selection Sort is that the inner loop only iterates as long as is necessary to find the insertion point.
- In the worst case, it will iterate over the entire sorted part. In this case, the number of iterations is the same as for Selection Sort; hence, the worst-case running time is $O(n^2)$- the same as Selection Sort and Bubble Sort.
- At the other extreme, however, if the array is already sorted, the inner loop won't need to iterate at all. In this case, the running time is $\Omega(n)$, which is the same as the running time of Bubble Sort on an array which is already sorted.
- Bubble Sort, Selection Sort and Insertion Sort are all in-place sorting algorithms.
- Bubble Sort and Insertion Sort are stable, whereas Selection Sort is unstable.

Criteria for choosing a sorting algorithm

Criteria	Sorting algorithm
Small number of items to be sorted	Insertion Sort
Items are mostly sorted already	Insertion Sort
Concerned about worst-case scenarios	Heap Sort
Interested in a good average-case behaviour	Quicksort
Items are drawn from a uniform dense universe	Bucket Sort
Desire to write as little code as possible	Insertion Sort
Stable sorting required	Merge Sort

Reference: Pollice G., Selkow S. and Heineman G. (2016). Algorithms in a Nutshell, 2nd Edition. O' Reilly.

Recap

- Bubble Sort, Selection Sort, and Insertion Sort are all $O(n^2)$ in the worst case
- It is possible to do much better than this with even with comparison-based sorts, as we will see in the next lecture
- From this lecture on simple $O(n^2)$ sorting algorithms:
  - Bubble Sort is extremely slow, and is of little practical use
  - Selection Sort is generally better than Bubble Sort
  - Selection Sort and Insertion Sort are "good enough" for small input instances
  - Insertion Sort is usually the fastest of the three. In fact, for small n (say 5 to 10 elements), Insertion Sort is usually faster than more complex algorithms