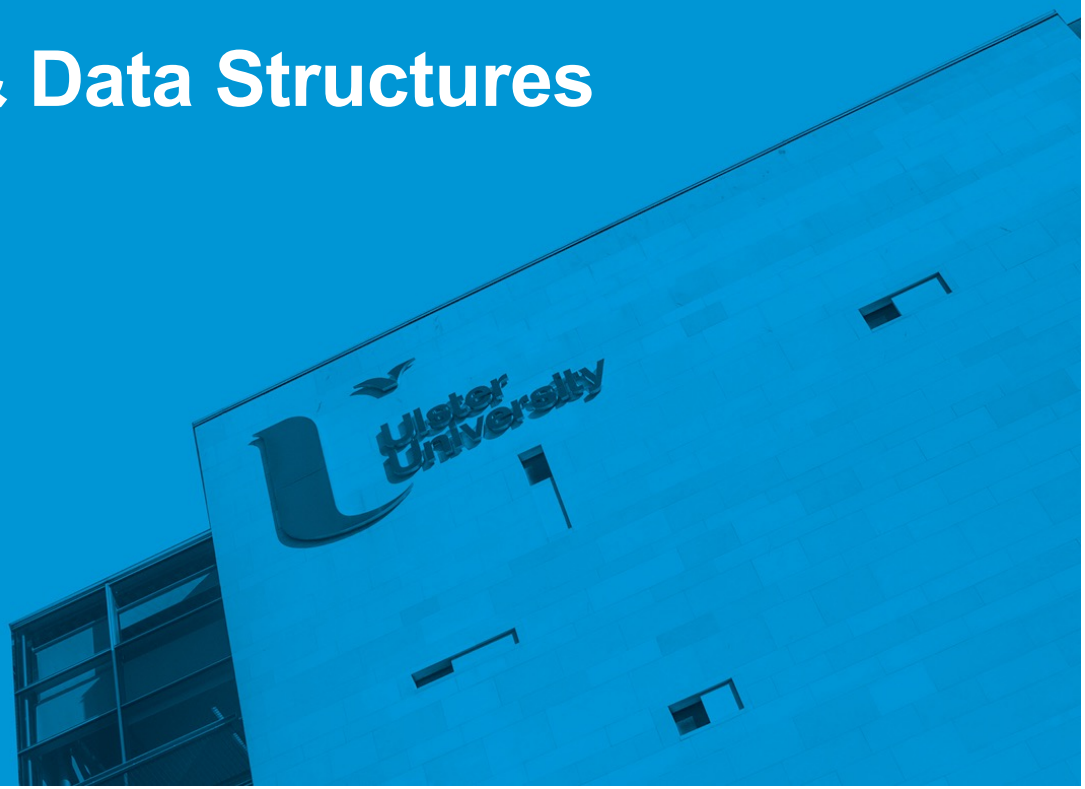
Ulster University

COM498 Algorithms & Data Structures

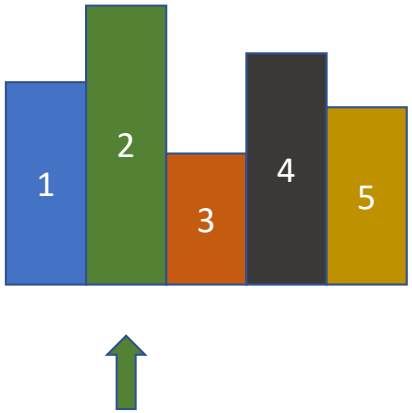
7.1 Insertion Sort



Insertion Sort

- Orders a list of values by repetitively inserting a particular value into a sorted subset of the list
    - Consider the first item to be a sorted sub-list of length 1
    - Insert the second item into the sorted sub-list, shifting the first item if needed
    - Insert the third item into the sorted sub-list, shifting the other items as needed
    - Repeat until all values have been inserted into their proper positions in the list

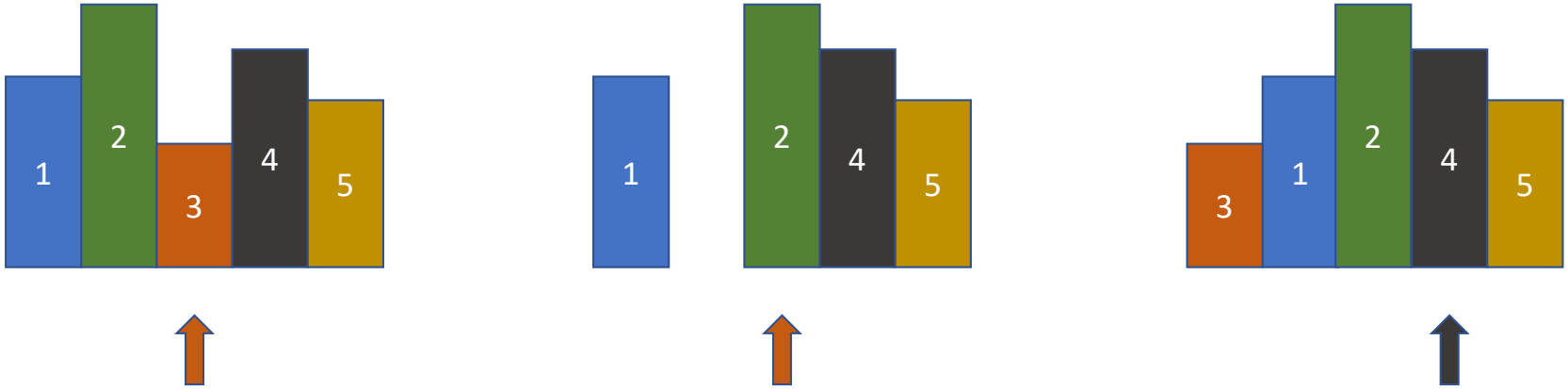
Insertion Sort



- Consider the second book:
  - If it is taller than the first book, then two of the books are sorted
  - If not, remove the second book, slide the first book to the right and insert the book that was removed (second book) into the first position on the bookshelf

Insertion Sort

- Consider the third book:
  - If it is taller than the second book, then three of the books are sorted
  - If not, remove the third book, slide the second book to the right
  - If the book that was removed is taller than the first book, then insert the book that was removed (third book) into the second position on the bookshelf
  - If not, then slide the first book to the right and insert the book that was removed (third book) into the first position on the shelf

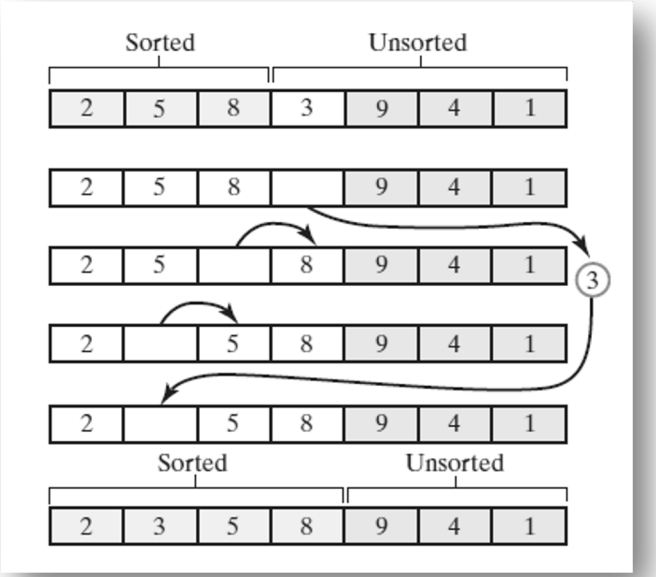


Insertion Sort

- Arrow scans from left to right
- Entries to the left (including the one at the arrow) are in ascending order
- Entries to the right still need to be sorted (an entry to the right of the arrow may be smaller than any entry to the left)

Iterative Insertion Sort

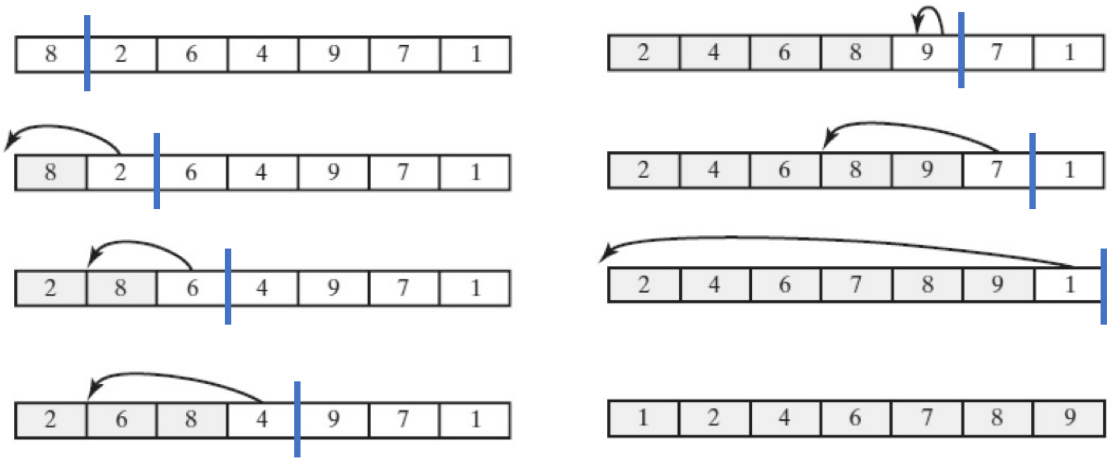
- Insertion sort of an array partitions the array into two parts: (1) sorted, (2) unsorted



- Sorted part initially contains just the first entry
- Unsorted part contains remaining entries
- Algorithm removes the first entry from the unsorted part and inserts it into its proper position in the sorted part
- Choose the proper position by comparing the unsorted entry with the sorted entries beginning at the end of the sorted part and continuing towards its beginning (shifting entries in the sorted part to make room for the insertion)

Iterative Insertion Sort

- Insertion sort of an array of integers – at each pass of the algorithm, the sorted part expands by one entry as the unsorted part shrinks by one entry (eventually the unsorted part is empty and array is sorted)



Iterative Insertion Sort

- Pseudocode of iterative algorithm for Insertion Sort:

```
Algorithm insertionSort(a)
// Sorts the entries of an array a.

loop i from first position + 1 to last position
    set nextToInsert to a[i]

    // find the gap
    set index to i - 1
    while index >= 0 and a[index] > nextToInsert
       set a[index + 1] to a[index]
       decrement index

    // insert value into the gap
    set a[index + 1] to nextToInsert
```

- Sorted part of the array initially contains one entry `a[first]`, so loop begins at index position `first + 1` to process the unsorted part

Scenario

- Implement the iterative version of the Insertion Sort
    - As for the previous sorts, add code to display the state of the array at every stage and to display the number of comparisons and array updates (changes to individual array elements).
    - Test the implementation for an integer array of 10 values where…
        1. the array to be sorted is in random order
        2. the array is already in sorted order
        3. the initial array is in reverse order

Recursive Insertion Sort

- Insertion Sort also has a natural recursive form:

```
Algorithm insertionSort(a, firstPos, lastPos)
// Sorts entries of an array a from firstPos through lastPos

if firstPos < lastPos
    call insertionSort(a, firstPos, lastPos – 1)
    set nextToInsert to a[lastPos]

    // find the gap
    set index to lastPos - 1
    while index >= 0 and a[index] > nextToInsert
       set a[index + 1] to a[index]
       decrement index

    // insert value into the gap
    set a[index + 1] to nextToInsert
```

- If you sort all but the last item in the array using an insertion sort (i.e. smaller problem), you can insert the last item into its proper position within the sorted array

Scenario

- Implement the recursive version of the Insertion Sort
    - As for the previous sorts, add code to display the state of the array at every stage and to display the number of comparisons and array updates (changes to individual array elements).
    - Test the implementation for an integer array of 10 values where…
        1. the array to be sorted is in random order
        2. the array is already in sorted order
        3. the initial array is in reverse order

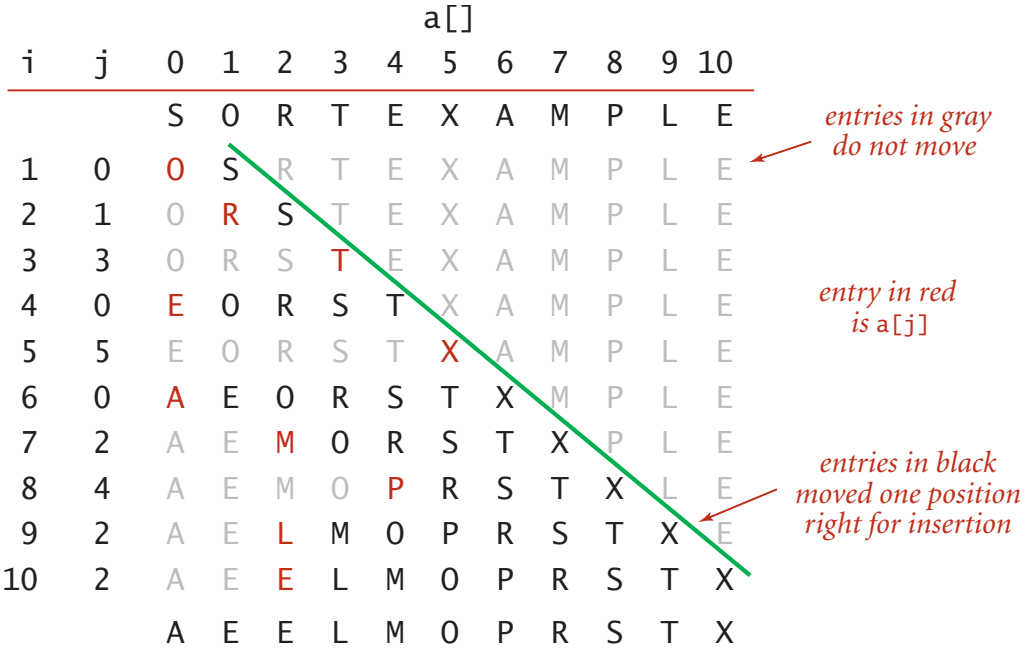
Efficiency of Insertion Sort

- Insertion sort uses ~ ¼ N^2 comparisons and ~ ¼ N^2 array updates on average

- Best case: if array is in ascending order, insertion sort makes $N - 1$ comparisons and N - 1 (really, 0) updates, so is O(n)

- Worst case: if array is in descending order (with no duplicates) insertion sort makes ~ ½ N^2 comparisons and ~ ½ N^2 updates, so is $O(n^2)$

- The closer an array is to sorted order, the less work insertion sort needs to do



Scenario

- Compare the performance of the iterative and recursive Insertion Sort techniques
  - Revisit the challenge from Section 6.3 and measure the execution time to sort 1000 arrays of 100, 200, 400, 800, 1600, 3200 and 6400 elements.
  - Perform the exercise for both the iterative and recursive versions of the Insertion Sort and report the results as a series of 3 values per line – array size, iterative sort time, recursive sort time.

Sorting within applications

Ulster University

- Examples so far have been within the Sort class, with demonstrations of sorting implemented in the `main()` method
    - A more realistic approach is to build an application that requires sorting as an enabler for key elements of functionality

Scenario

- **Cardball** is a sports simulation game in which each player is dealt 5 playing cards from a shuffled pack. To play the game, each player is required to sort the cards in their hand and play them in descending order of value. After each player has played a card, the player who played the card with the highest value is awarded a goal. If more than one player plays the highest valued card, then no goal is scored in that round.

- Revisit the Card project and add a class `Deck` that represents a standard deck of playing cards (represented by the `Card` class already developed). The `Deck` class requires methods to

    1. Create a new deck of 52 (different) `Card` objects in a random order
    2. Deal a card from the deck

- Develop a class `Cardball` to play an automated 2-player game (i.e. one with no user input) according to the rules above

    - The game should run in the `main()` method, while the class can contain any supporting variables or methods that are required.