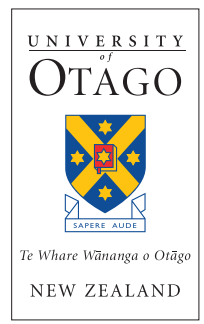
# Cosc 241
# Programming and Problem Solving
# Lecture 16 (18/4/2019)
# Insertion and Selection Sort

Lech Szymanski
lechszym@cs.otago.ac.nz

*Keywords*: sorting, selection sort, insertion sort



# Background

An entire volume of Knuth's *The Art of Computer Programming* is devoted to sorting and searching. Why?

- 'Sorting algorithms are prevalent in introductory computer science classes, where the abundance of algorithms for the problem provides a gentle introduction to a variety of core algorithm concepts, such as big O notation, divide and conquer algorithms, data structures, randomized algorithms, best, worst and average case analysis, time-space tradeoffs, and lower bounds.' (wikipedia)
- '. . . over 25 percent of the running time . . . spent on sorting, . . . many installations in which sorting uses more than half of the computing time.' (Knuth)
- Most day to day sorting problems are 'solved' but there are still interesting variations both theoretically and practically, particularly for search purposes.

# Why all the algorithms?

- The number of available sorting algorithms is enormous. Why?
- Simplicity v speed. Simple can beat ‘asymptotically fast’ on small amounts of data.
- Memory use – how much extra storage is needed beyond that used for the actual data?
- Adaptability – does the algorithm improve with partially presorted or otherwise structured data?
- Worst case as opposed to average case behaviour and knowing when it matters!

# Sorter interface

```
public interface IntSorter {

  public void sort(int[] a);

  /* ... */
}
```

# Selection sort

Find the smallest item in the list. Swap it with the first item. Find the smallest item in the remainder of the list, swap it with the first item of the remainder. Repeat until finished.

We can think of this as a recursive algorithm using subarrays. The needed ingredients are just a method that returns the position of the minimum element of a subarray, and a method that swaps elements in two positions.

# Swap

You know how to do this!

```
public class ArrayManipulation {
  /* ... */

  public static void swap(int[] a, int i, int j) {
    int t = a[i];
    a[i] = a[j];
    a[j] = t;
  }

  /* ... */
}
```

# Selection sort

```
public class SelectionSort implements IntSorter {
  /* ... */

  public void sort(int[] a) {
    sort(a, 0, a.length);
  }

  private void sort(int[] a, int left, int right) {
    if (left >= right) {
      return;
    }
    ArrayManipulation.swap(a, left, minPosition(a, left, right
    sort(a, left + 1, right);
  }

  /* ... */
}
```

# Minimum position

```
public class SelectionSort implements IntSorter {
  /* ... */

  private int minPosition(int[] a, int left, int right) {
    int pos = left;
    int minValue = a[left];
    for (int i = left + 1; i < right; i++) {
      if (a[i] < minValue) {
        pos = i;
        minValue = a[i];
      }
    }
    return pos;
  }

  /* ... */
}
```

# Selection sort analysis

- Extra space required: $O(1)$ (for the location of the minimum and the swap).
- We always examine the whole remaining list to find the minimum position, so do $n - 1$ comparisons in the first pass, $n - 2$ in the second etc.
- So time required is *always* $O(n^2)$ (there is no real worst case/best case since we always examine the whole remaining list).

# Truth in advertising

The recursive code for selection sort chokes on arrays of any decent size (because the call stack capacity is limited). The recursion is easy to eliminate in this case:

```
public class SelectionSort implements IntSorter {
  /* ... */

  public void sort(int[] a, int left, int right) {
    while (left < right) {
      swap(a, left, minPosition(a, left, right));
      left++;
    }
  }

  /* ... */
}
```

# Insertion sort

Process the array one element at a time. Insert each new element into its correct position among the previously processed elements.

Observation suggests that this is the method most people use when sorting a small stack of exam papers (a minority prefer selection sort).

# Naive insertion sort

```
public class NaiveInsertionSort implements IntSorter {
  /* ... */

  public void sort(int[] a) {
    for (int right = 1; right < a.length; right++) {
      int pos = findPosition(a, right);
      ArrayManipulation.insert(a, pos, right + 1, a[right]);
    }
  }

  private int findPosition(int[] a, int right) {
    int pos = 0;
    while (pos < right && a[pos] < a[right]) {
      pos++;
    }
    return pos;
  }

  /* ... */
}
```

# Insert

```
public class ArrayManipulation {
  /* ... */

  public static void insert(int[] a, int index,
                            int right, int value) {
    for (int dest = right - 1; dest > index; dest--) {
      a[dest] = a[dest - 1];
    }
    a[index] = value;
  }

  /* ... */
}
```

# Insertion sort analysis

- Extra space required: $O(1)$ – for the insertion position and the insertion itself.
- We always search up to the position we insert at, and then insert from there until the end of the current subarray.
- So time required is *always* $O(n^2)$ (there is no real worst case/best case).
- We can do better by combining the *find position* and *insert* steps.

# Better insertion sort

- When the time comes to insert the element from position `i` the previous elements are sorted and those greater than `a[i]` need to be moved forward.
- So, read backwards from position `i`, moving elements forward until the necessary hole is created into which we can place the value `a[i]` (which we need to store in advance since it will be written over in the first step).
- Notice this works really well if `a` is nearly sorted – most items will move only a short distance (or not at all) and performance will be more like $O(n)$ than $O(n^2)$.

# Better insertion sort

```
public class InsertionSort implements IntSorter {
  /* ... */

  public void sort(int[] a) {
    for (int i = 0; i < a.length; i++) {
      findAndInsert(a, i, a[i]);
    }
  }

  private void findAndInsert(int[] a, int index, int value) {
    index--;
    while (index >= 0 && a[index] > value) {
      a[index + 1] = a[index];
      index--;
    }
    a[index + 1] = value;
  }

  /* ... */
}
```

# Experiments

- The main reason to use an interface was to allow us to plug in various classes that implement `IntSorter` into a test harness.
- That allows us to check that they are correct, and also to compare their speeds.
- We can also write a wrapper around `Arrays.sort` in order to compare with the system provided sort routines.
- Obviously there is room for improvement.