Simple Sorting

Bryn Mawr College
CS206 Intro to Data Structures

Bubble Sort

• Start at the left end of the array and compare the first two elements.
• If the one on the right is bigger, you don’t do anything; otherwise, swap them.
• Move one position right.
### Bubble Sort - Pseudocode

```pseudocode
BubbleSort( int a[])
    int out, in
    for out = a.length-1 to 1  //outer loop backward
        for in = 0 to out   //inner loop forward
            if a[in] > a[in+1]
                temp = a[in]
                a[in] = a[in+1]
                a[in+1] = temp
        end for
    end for
```

### Bubble Sort – Invariant & Complexity

- Invariants: conditions that remain unchanged as the algorithm proceeds.
- Invariant for bubble sort algorithm:
  - the data items to the right of out are sorted.
- Complexity:
  - N-1 comparisons on the first pass,
  - N-2 on the second, and so on.
  - \( (N-1) + (N-2) + (N-3) + \ldots + 1 = N(N-1)/2 \)
  - \( O(N^2) \)
Selection Sort

- Start at position 0, making a pass through all the items and picking (or selecting, hence the name of the sort) the smallest one.
- This smallest item is then swapped with the item on the left end of the line, at position 0.
- Now the leftmost item is sorted and won’t need to be moved again.
- Start at position 1, finding the minimum, and swap with position 1.
- …

Selection Sort - Pseudocode

```pseudocode
SelectionSort(int A)
    n=A.length
    for i = 0 to n – 1
        min=i
        for j=i+1 to n - 1
                min=j
        end for
        swap A[i] and A[min]
    end for
```
Selection Sort – Invariant & Complexity

• Invariant:
  o The data items with indices less than or equal to $i$ are always sorted.
• Complexity:
  o $N^2(N-1)/2$
  o $O(N^2)$
• Selection sort vs. bubble sort:
  o $O(N^2)$ comparison performed
  o $O(N)$ swaps

Insertion Sort

• Idea:
  for $i = 1$ to $n - 1$:
    insert $A[i]$ in its proper place amongst $A[0..i]$.
• Example
  o $4 \ 3 \ 2 \ 1 \rightarrow 3 \ 4 \ 2 \ 1 \rightarrow 2 \ 3 \ 4 \ 1 \rightarrow 1 \ 2 \ 3 \ 4$
• Note that inserting $A[i]$ in the correct place implies shifting up some, maybe all, elements of $A[0..i-1]$ to make room for $A[i]$. 
Insertion Sort – Pseudocode

```plaintext
InsertionSort(int A)
    n = A.length
    for out = 1 to n – 1
        temp = A[out]
        in = out
        while in > 0 and A[in-1] >= temp
            in--
        end while
        A[in] = temp
    end for
```

Insertion Sort – Invariant & Complexity

- Invariant:
  - At the start of iteration $i$ of the outer loop, subarray $A[0..i-1]$ consists of the elements originally in $A[0..i-1]$, but in sorted order.
  - At termination, $i=n$, so the invariant implies that subarray $A[0..n-1]$ (i.e., the whole array) consists of the elements originally in $A[0..n-1]$, but in sorted order. (correctness of insertion sort)

- Complexity
  - Outer loop: $n$ iterations
  - Inner loop: $\leq n$ iterations
  - Per iteration: constant amount of work
  - $O(N^2)$
Generics

• Generics enable types (classes and interfaces) to be parameters when defining classes, interfaces and methods.
• Benefits over non-generic code
  o Stronger type checks at compile time.
  o Elimination of casts.
    • List list = new ArrayList();
      list.add("hello");
      String s = (String) list.get(0);
    • List<String> list = new ArrayList<String>();
      list.add("hello");
      String s = list.get(0); // no cast
  o Enabling programmers to implement generic algorithms.

Generic Classes

• To define a generic class:

  Type definition block defines variable E

  public class ArrayList<E> extends AbstractList<E>
  implements List<E>, RandomAccess, Cloneable, Serializable

• To use a generic class:

  ArrayList<Dog> arr = new ArrayList<Dog>();
Defining Generic Classes and Interfaces

```java
public class Pair<E> {
    private E first;
    private E second;
    public Pair(E first, E second) {
        this.first = first;
        this.second = second;
    }
}
```

we can use the Pair class by providing a concrete type:

- Dog d = new Dog("Rolf", null);
- Retriever r = new Retriever("Clover", null);
- Pair<Pet> petPair = new Pair<Pet>(d, r);

```java
public interface Comparator<T>
{
    int compare(T lhs, T rhs);
}
```

Scope of Type Variables

- You can use a type variable throughout the class (including in the class declaration itself).
- You **cannot** refer to it in a static method or declare static variables of that type:

```java
public class MyClass<T> implements Iterable<T> {
    private T foo;  //OK

    // Compile error: “Cannot make a static
    // reference to the non-static type T”
    private static T bar;
}
```
**Historical Notes**

- Before generic types were introduced, a class such as `ArrayList` always just stored type `Object`. You could write dangerous code like this:

```java
ArrayList arr = new ArrayList();
arr.add(new Dog("Rolf"));
arr.add(new Retriever("Clover"));
arr.add(new Volkswagen());
for (int i = 0; i < arr.size(); ++i)
    Dog d = (Dog) arr.get(i); // need downcast
d.speak();
```

This code compiles, but gives you `(ClassCastException)` at runtime.

**Historical Notes (cont.)**

As of 1.5, `ArrayList` is a generic class. Thus, in the next example, the compiler ensures that anything we add() to the list is compatible with the type `Dog`:

```java
ArrayList<Dog> arr = new ArrayList<Dog>();
arr.add(new Dog("Rolf"));
arr.add(new Retriever("Clover"));
// arr.add(new Volkswagen()); // Error is caught at compile time
for (int i = 0; i < arr.size(); ++i) {
    Dog d = arr.get(i); // no downcast
d.speak();
}
```

You can still use `ArrayList` the old way, but it is called a “raw type” and you get compiler warnings.