COMP 2210 – Exam 1 Study Guide Robert Sanek

**GENERALITY**

**Equality**

* All classes inherit equals() method; default implementation is the same as == (aliasing/object identity).
* Override equals() with logical equality.
* Equals method implements an **equivalence relation** on non-null object references:
  1. **Reflexive**: x.equals(x) is true
  2. **Symmetric**: x.equals(y) iff y.equals(x)
  3. **Transitive**: if x.equals(y) && y.equals(z) then x.equals(z)
  4. Non-Null: x.equals(null) is false
* The equals() Recipe:

public Boolean equals(Object obj)

if (obj == this) return true;

if (obj == null) return false;

if (obj.getClass() != this.getClass()) return false;

Book that = (Book) obj;

return (logic);

1. Check for aliasing
2. Check for null
3. Check for type compatibility
4. Cast from Object
5. Implement your actual logical equality

* The **Comparable** Interface – comparing two things & deciding equal to, less than & greater than. compareTo() is the only method in the interface.
  + x.compareTo(y) implementation:
    - x **equal to** y return **zero**
    - x **less than** y return **negative** integer
    - x **greater than** y return **positive** integer
  + compareTo() implements a **total order** on non-null object references. (throws exception for null refs or incompatible types)

**Generics** allow a type variable to be used in place of a specific type name.

* All variables must be declared before they are used. Use a **type variable** to specify the type and target.
  + public static <T> int search(T[] a, T target) {}
  + When calling search(), clients will preface the method name with the class name: SearchLib.<String>search(array, “string”);
* Type variables can’t be instantiated; they don’t exist at runtime.
* **Comparable** interface defines the abstract behavior of one object comparing itself to another object.
  + a.compareTo(b) – a is comparing itself to target
* **Comparator** interface defines the abstract behavior of a third party comparing two objects to each other compare()
  + c.compare(a, b) – c is an instance of a class that implements the Comparator interface. c compares a and b.
  + public int search(Anything[] a, Anything target, Comparator c) – c should know how to compare two Anythings to each other.
* compare() and compareTo() have the same return values.
* public <T> int search(List<T> a, T target) {}

An **Iterator** is an object that allows a programmer to traverse through all the elements of a collection, regardless of specific implementation.

* Iterator<T> itr = a.iterator(); | boolean hasNext() {} | E next() {} | void remove() {}
* Allows traversals regardless of implementation & can provide much faster access than get() method.

**ALGORITHM ANALYSIS**

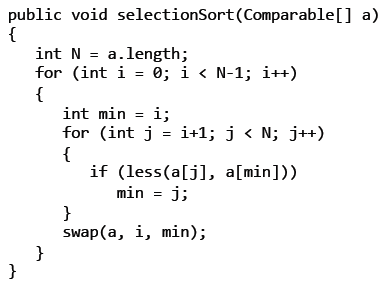
Empirical & Mathematical **algorithm analysis**

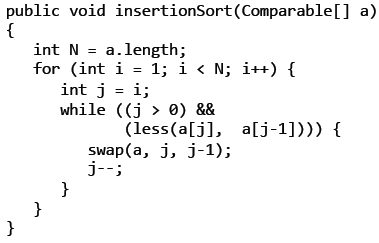
* Empirical – analyze running time based on observations and experiments. (Table of N, elapsed time, & ratio)
* **Mathematical** – develop a cost model that includes cost for individual operations.

**Big-Oh notation** – a function that describes the running time in terms of the problem size (n), providing an upper bound on running time

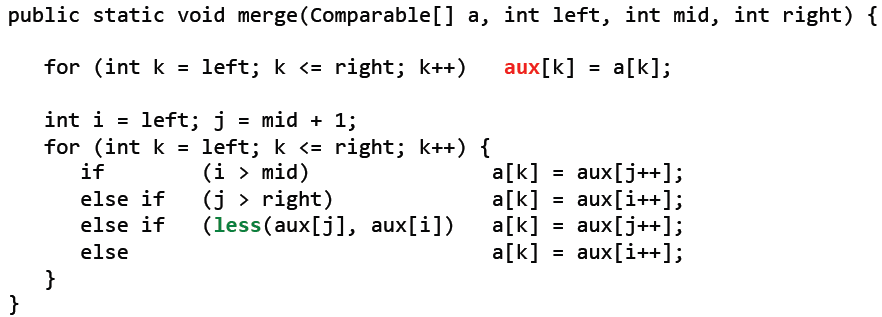
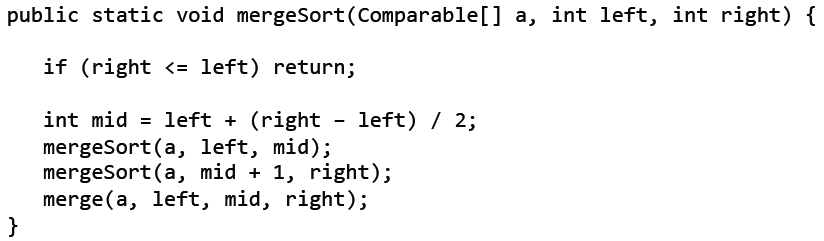
* Dominant term of time complexity function is used in Big-Oh: all lower-order terms & coefficients are ignored.
* **Common orders of growth**, sorted: 1, log N, N, N\*log N, N2, N3, 2N
* Advances in hardware cannot compensate for some big-oh time complexities; “Software Progress Beats Moore’s Law”
* Best-case, **worst-case**, average-case.
* **Calculating worst-case Big-Oh**
  1. All simple statements & primitive operations have constant cost.
  2. The cost of a sequence of statements is the sum of the costs of each individual statement.
  3. The cost of a selection statement is the cost of the most expensive branch.
  4. The cost of a loop is the cost of the body multiplied by the maximum number of iterations that the loop makes.
* Constant halving: log2 N cost

**SORTING**

**Selection Sort** [O(N2), regardless of input] – Walk from left to right, select correct element and exchange it with current one. On kth iteration, first k elements are in correct, sorted position and k+1 elements are all greater than or equal to first k elements.

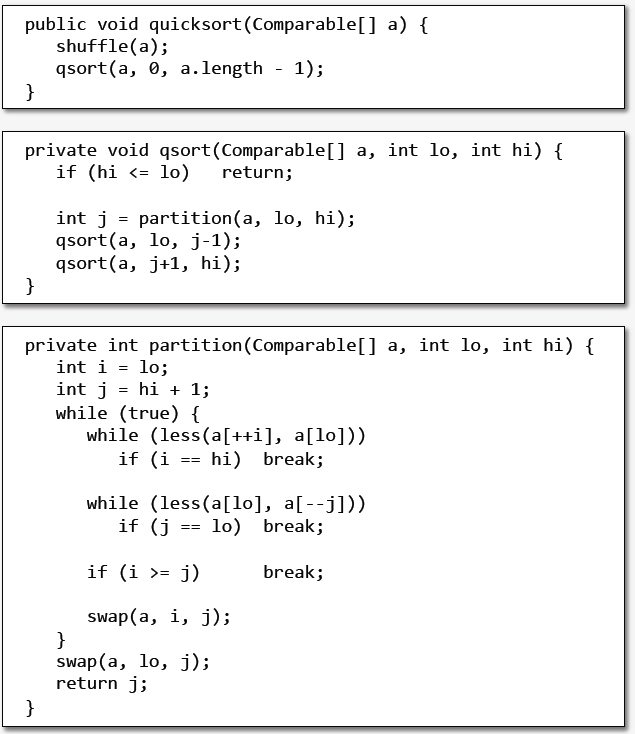


**Insertion Sort** [O(N2), varies if sorted/not] – Walk from left to right, insert element in current location in sorted order to its left. After k steps, first k+1 elements are sorted relative to each other, elements k+2 through last element are in original order.



**Merge Sort** [O(N\*log N)] – Divide/Conquer/Combine: Divide array in half, sort each half recursively, combine sorted halves into single array. Asymptotically optimal, but requires O(N) extra space.

(mid will be lower integer if not whole.)

**QuickSort** [O(N\*log N), average case; O(N2), worst case] – Pivot/Partition/Sort: Select a pivot, partition array, sort each partition recursively. The “**sort-of-choice**” for many situations, since it’s expected to be faster on typical data sets. Asymptotically optimal for comparison sorting, but only in average case.

Choosing a pivot value: choose randomly.

Sort Methods in Java, (1998-2011):

* Collections.sort(): mergesort
* Arrays.sort(): quicksort (primitives), mergesort (references)

(2011-present):

* Collections.sort(): mergesort/TimSort
* Arrays.sort(): quicksort (primitives), mergesort/TimSort (references)

For **comparison sorts**, N\*log N is lower bound for number of comparisons necessary.

