Com S 228
Fall 2010
Programming Assignment 2
Due at 11:59 pm Friday, October 8

This assignment is to be done on your own. If you need help, see me or one of the TAs. Please make sure you understand the "Academic dishonesty" section of the syllabus.

Please start the assignment as soon as possible and get your questions answered *early*.
Read through this specification *completely* before you start.

Remember that the exam on 9/27 will include generic types, comparators, and quicksort, so getting started on this homework is a also a good way to study these topics for the exam. There is a Getting Started section near the end of this document.

WebCT

The WebCT discussion for Assignment 2 is a good place to post general questions. Please do not post or attach any source code for the assignment. Also check the "official clarification" thread for corrections, hints, and answers to common questions.

Introduction

The purposes of this assignment are

1. to practice working with generic types and generic methods in Java
2. to become familiar with the Comparable and Comparator interfaces
3. to implement and debug two variations of the quicksort algorithm
4. to investigate the behavior of quicksort when aspects of the partitioning strategy are varied

Summary of tasks

The tasks to be completed by you are summarized below:

1. Complete the implementation of the QuickSorter class, including the partition methods described in detail in the next section
2. Complete the implementation of the SortUtil class that contains the insertion sort algorithm
3. Implement the MedianStrategy class
4. Implement the SortMain class containing the main() method

The sample code for this assignment also includes classes BasicStrategy and ComparableQuickSorter, which are fully implemented.

Details

The QuickSorter<T> class

This class encapsulates the implementation of the quicksort algorithm along with various configuration options and state variables used for instrumenting the code. To use a QuickSorter, you instantiate one for your data type, configure it as you wish, and invoke sort(). An example of using the QuickSorter to sort an array of Foo objects might look something like this:

```
Foo[] arr = ...
Comparator<Foo> myComp = ...
QuickSorter<Foo> sorter = new QuickSorter<Foo>(myComp);
sorter.setUseThreeWay(true);
sorter.sort(arr);
```

Configuration parameters

The configuration parameters consist of the following (which will be instance variables in your QuickSorter class):

- A Comparator, which must be provided in the constructor
- An instance of IPivotStrategy, which may be optionally provided in the constructor. If none is provided, the algorithm should use an instance of BasicStrategy. The IPivotStrategy provides two things: a strategy for selecting the pivot value, and a minimum length. Any subarray that is shorter than the minimum length should be sorted using insertion sort.
- A boolean flag useThreeWayPartition indicating whether to use *normal partitioning* or *three-way partitioning.* This value is set using the method setUseThreeWay()

Implementation of quicksort

Your quicksort algorithm should be written to perform the partitioning step using private methods for partitioning as specified below. If useThreeWayPartition is false, call the partitionNormal method, and if useThreeWayPartition is true, call the partitionThreeWay method.

Prior to calling partition, your algorithm checks whether the length of the subarray is less than the value returned by calling minLength() on the current instance of IPivotStrategy. If so, the subarray is sorted using insertion sort. Within the partition method, the pivot is selected using the indexOfPivotElement() method of the strategy.

Normal partitioning

For normal partitioning, you *must* adapt either of the following algorithms:

a) the algorithm on p. 425 of the text (QuickSortExample.java, posted for 9/15, contains a sample implementation)

b) the partition algorithm in QuickSortCode.java, posted for 9/15 (you'll need to separate the partitioning step from the rest of the quicksort code)

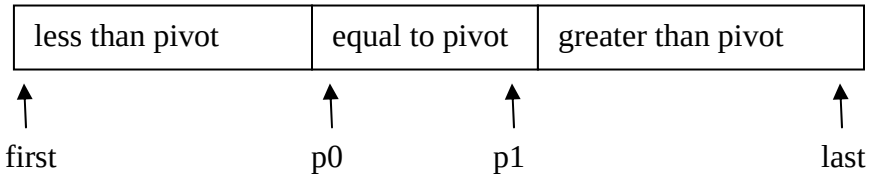
Implement the partition step in a method **`partitionNormal`** as specified below. (Note: normally such a method would not be public. We are asking you to make it public to simplify testing.)

precondition: (last – first + 1) >= minLength as returned by pivot strategy
postcondition:
arr[p] is equal to the pivot value,
all elements left of p are <= pivot, and
all elements right of p are >= pivot,
where p is the return value.

```
public int partitionNormal(T[] arr, int first, int last)
```

Three-way partitioning

The purpose of using three-way partitioning is to enable the quicksort algorithm to deal effectively with data sets having many duplicate elements. The idea is that in the partitioning step, *all* elements equal to the pivot value are moved into a "block" in the middle of the subarray.



The recursive calls now only need to sort the subarray first through p0 – 1 and the subarray p1 + 1 through last. The only change in the recursive part of the algorithm is that the partition step now returns a pair (p0, p1).

Implement the three-way partition algorithm in a method **`partitionThreeWay`** as specified below. (Note: normally such a method would not be public. We are asking you to make it public to simplify testing.)
:

precondition: (last – first + 1) >= minLength as returned by pivot strategy
postcondition:
p0 <= p1,
arr[p0] through arr[p1] are equal to the pivot value,
all elements left of p0 are < pivot, and
all elements right of p1 are > pivot.
where (p0, p1) is the return value.

```
public Pair<Integer> partitionThreeWay(T[] arr, int first, int last)
```

For three-way partitioning, you must implement the following algorithm:

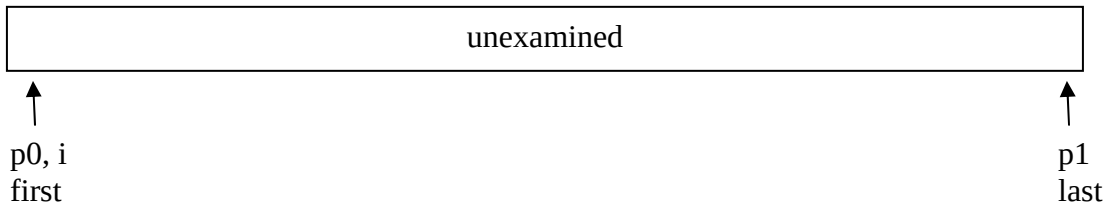
Initially, select a pivot value and set p0 = first, i = first, and p1 = last.
Then execute the following loop:

while i is less than or equal to p1
if arr[i] is less than the pivot, swap it into position p0, increment p0 and i
else if arr[i] is greater than the pivot, swap it into position p1, increment p1
else increment i

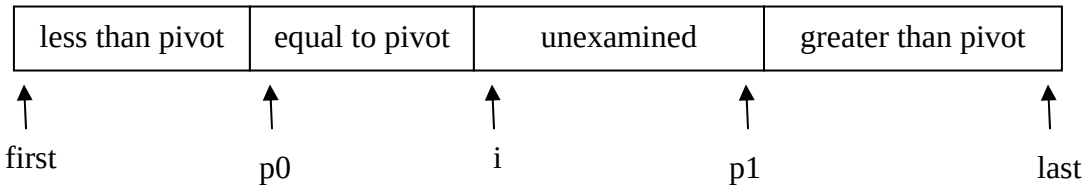
The key to the algorithm is the following loop invariant. Note that the invariant is true when p0, p1, and i are first initialized, and (if correctly implemented) remains true after each iteration.

all elements to the left of arr[p0] are < pivot
all elements arr[p0] through arr[i - 1] are equal to pivot
all elements to the right of arr[p1] are > pivot
elements arr[i] through arr[p1] are unexamined

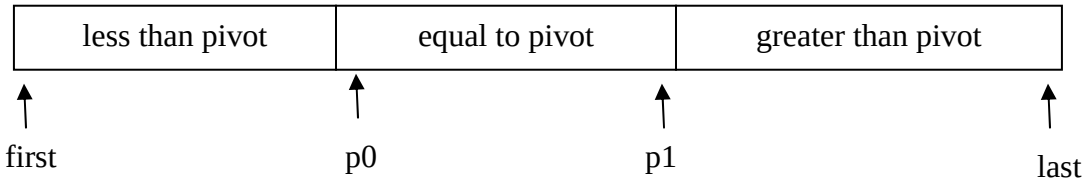
Initial state:



During execution:



On termination, since the invariant is still true when i > p1, we have the following:



Instrumentation

The following information should be available after sorting is complete:

- Total number of comparisons of array elements
- Total number of swaps of array elements
- Maximum depth of the call stack for recursive calls to quicksort

These values should always be reset to zero each time sort() is called.

The quicksort algorithm often calls the insertion sort algorithm in the SortUtil class (described below), and the comparisons and swaps performed in these calls to insertion sort need to be counted in the stats. The insertion sort method returns the number of comparisons performed. Insertion sort does not explicitly rely on swapping elements, but a swap is roughly equivalent to three assignments, and we can estimate the number of element assignment operations in a given call to insertion sort and then divide by 3:

equivalent swaps = (length of subarray – 1 + number of comparisons) / 3

So when your quicksort calls insertion sort, add in the number of comparisons (as returned by the insertion sort method) to your comparison counter, and add the above value to the counter for total swaps.

The SortUtil class

This class contains static methods for performing insertion sort. Note that the insertion sort code should only be implemented *once* in this class. The less general methods should delegate to the more general versions of the method. For example, the sort method for a Comparable type can simply call the corresponding method that takes a Comparator, since you can easily make a Comparator from a Comparable type T using the idiom

```
Comparator<T> comp = new Comparator<T>()
{
  @Override
  public int compare(T lhs, T rhs)
  {
    return lhs.compareTo(rhs);
  }
};
```

(See also the findMin method in the class GenericsTest, posted in the sample code for 9/17, for an example of this technique.)

Unlike the QuickSorter class, the SortUtil class is not generic and is completely stateless (has no instance variables) It uses generic methods to make the sorting algorithm available for arbitrary types. In order to support the instrumentation requirements for QuickSorter, the most general version of the insertion sort methods has a return type of int. It should return the total number of comparisons of array elements performed. (The number of swaps is deduced as described above in the discussion of instrumentation for QuickSorter.)

The MedianStrategy<T> class

Create a new implementation of IPivotStrategy called MedianStrategy that selects the pivot by randomly choosing *k* indices from the subarray and returning the index containing the median of the *k* elements, where *k* is a parameter provided to the constructor. (For example, when *k* = 1, the strategy just returns a randomly selected index.) Recall that the *median* is just a "middle" value in a set, that is, a value *M* with the property that half the elements are <= *M* and half the elements are >= *M*.

You need to provide three constructors:

> 1) A constructor with one int parameter, the value *k*. The constructor should throw IllegalArgumentException if the argument is less than 1 or is an even number. The constructor should create a new instance of Random without specifying a seed. The minLength() method should return *k*.
>
> 2) A constructor as above with a second int parameter that represents the value to be returned by minLength(). The constructor should throw IllegalArgumentException if the second argument is less than *k*.
>
> 3) A constructor as in (2) with a third parameter of type long, allowing the caller to specify the seed for the Random instance.

The "random" elements must be chosen using the nextInt(int) method of java.util.Random, using an instance of Random that is created upon construction of the MedianStrategy instance. Use the Random instance to select *k* indices between first and last, inclusive. It is *not* necessary to ensure that the indices are distinct.

Remember that you are not finding the median of the selected indices, but the index of the median array element for the selected indices. (One relatively easy technique: put the selected indices into an array of Integer, and sort them using insertion sort using a custom Comparator that orders indices i, j by comparing elements arr[i], arr[j] according to the original comparator with which you're sorting.)

You do not have to get carried away making the implementation as efficient as possible, since it is reasonable to assume that *k* will always be small (certainly less than 10 in practice).

The main class SortMain

This class includes a main() method. Your main method will should allow you to test and compare various sorting options by sorting arrays of Integers. There are three parameters:

> size – number of elements in the test array
> range – values to be generated for the test array are in range 0 through range - 1
> seed – seed for the random number generator for creating the test arrays

The idea is that if range is smaller than size, the generated array will have lots of duplicates; if range is close to size, the array will end up with few duplicates. The purpose of the seed to initialize the random number generator such that tests can be repeated with the same values. (An instance of Random with the same seed will always generate the same sequence of values).

These three values should be provided to main(String[] args) method as **command-line arguments**, separated by spaces, in the order listed above. If the wrong number of arguments is supplied, or if any of the arguments cannot be parsed as an integer, print a brief usage hint and return.

Your main method should run the following tests. Each step should be performed twice. For each step, measure the elapsed time in milliseconds for performing the sort (do not include other activities such as generating the array or creating the sorter object). Print a line of text including the pivot strategy being used, whether three-way partitioning is being used, the elapsed time in milliseconds, the number of comparisons, the number of swaps, and the maximum depth of recursion. To measure time you can use System.currentTimeMillis(), e.g.,

```
long start = System.currentTimeMillis();
doInterestingStuff();
long elapsed = System.currentTimeMillis() – start;
```

Each call to sort() should be surrounded by a try/catch block for StackOverflowError. If a StackOverflowError is caught, display that fact and show the maximum depth of recursion. (Do not print the stack trace!)

Item 2 should be performed using a QuickSorter constructed with a custom comparator to sort largest to smallest. For all others use a ComparableQuickSorter.

1. Create an instance *rand* of a random number generator (java.util.Random) using the given seed.
   Generate an Integer array of length *size* containing random values in the range 0 through range – 1
   Sort using BasicStrategy and normal partitioning.
2. Reseed the generator (using setSeed) and regenerate the array (*this should create an array identical to the first one*).
   Sort the array in reverse using BasicStrategy and normal partitioning.
3. Reseed the generator and regenerate the array.
   Reseed the generator again.
   Sort using MedianStrategy(1, 10, rand) and normal partitioning.
4. Reseed the generator and regenerate the array.
   Sort using BasicStrategy and three-way partitioning.
5. Reseed the generator and regenerate the array.
   Reseed the generator again.
   Sort using MedianStrategy(1, 10, rand) and three-way partitioning.
6. Sort (the already-sorted array) using BasicStrategy and normal partitioning
7. Sort (the already-sorted array) using BasicStrategy and three-way partitioning.
8. Reseed the generator.
   Sort (the already-sorted array) using MedianStrategy(3, 10, rand) and normal partitioning

Note

- No method other than main should ever produce output.
- Your main method should not read input interactively from the user
- Create helper methods as needed in your SortMain class to avoid code duplication (e.g. for generating a new array)
- Since it is your responsibility to fully test your code, you may wish to add a step to the process above to verify that the resulting array is actually sorted

- Remember that we do not allow calls to System.exit(). If you need to exit the main method early, use a return statement.

About command-line arguments

The purpose of the String[]argument in **`void main(String[] args)`** is to allow data to be passed into the runtime when it starts up. This data is available to your main method by examining the args array. If you are running from Eclipse, you edit the "Run Configuration" for the project (after running the main class at least once you can find the run configuration under the Run menu): select the Arguments tab and enter the arguments in the "Program arguments" pane. For example, to test with a one-million element array containing values 0 through 9999 using a seed of 42, your arguments would be

```
1000000  10000  42
```

When main starts, the string "1000000" will be in args[0], "10000" will be in args[1], and so on.

When running a command prompt, the arguments appear after the main class name. For example to run your main method from a command shell you could navigate to the project directory and enter something like:

```
java –classpath bin  edu.iastate.cs228.hw2.SortMain  1000000 10000 42
```

Getting started

There are really two distinct aspects to this assignment: getting used to generic types and comparators, and developing the partitioning algorithms. You can start on the partitioning and quicksort algorithms without using generic types at all, and you can work independently on SortUtil as a way to get familiar with generic types and comparators.

To get started with SortUtil, note that an example of the insertion sort algorithm can be found on p. 458 of the text, if you don't already have it in your notes. You will just have to adapt it to use a Comparator and then port it into the SortUtil class. Test using arrays of Integer or String. Try using a custom Comparator.

Independent of the above, you can work on your partitioning and quicksort algorithms. I would strongly recommend that you develop and debug your partitioning and quicksort code using arrays of ints first. It is probably simplest to start with normal partitioning. The 3-way partitioning algorithm can be developed independently. Step carefully through your code, using a debugger or println statements, and make sure that every index is exactly where it is supposed to be at each step. Be sure to try arrays with and without duplicates, already-sorted and random arrays, arrays of size 1, etc. Once you know your algorithm is working, it is not difficult to convert it to sort objects using comparators.

The pivot selection strategy MedianStrategy can also be developed and tested independently. That is, to start out, your quicksort can just use the first element of the subarray as a pivot, as in the examples. Then, adapt your code to use a BasicStrategy object. Then try integrating your MedianStrategy.

Documentation and style

Up to 15% of the points will be for documentation and style. See the style guidelines posted on WebCT.

What to turn in

All of your code should be added to the package edu.iastate.cs228.hw2. Submit a zip file, containing your source code only, in the correct package structure. The top-level directory should be edu.

See the submission HOWTO on the WebCT main page if you not certain how to do this. You **will** lose points for submission errors (and in the worst case, if you do not correct your submission errors before the late deadline, you may receive no credit at all for the assignment).

Late Penalties

Assignments may be submitted up to 24 hours after the deadline with a 25% penalty (not counting weekends/holidays). No credit will be given for assignments submitted after that time.