COMP 2140 Assignment 1: Sorting and Recursion

Helen Cameron

Due: Wednesday 12 October 2016 at 10:00 p.m.

Important Announcement

Until you agree to the honesty declaration (see the instructions at the end of this assignment under "Honesty Declaration"), you cannot hand in any assignments — that is, until you agree to the honesty declaration, you will not be able to see the assignment dropboxes in UM Learn.

How to Get Help

Your course instructor is helpful: Email me at Helen.Cameron@cs.umanitoba.ca or come to my office hours at 2:30–3:00 p.m. Mon/Wed/Fri or 4:00–4:30 p.m. Tues/Thurs in E2 477 EITC (or by appointment).

For email, please remember to put "[comp2140]" in the subject and use a meaningful subject, and to send from your UofM email account.

Course discussion groups on UM Learn: A discussion group for this assignment is available in the COMP 2140 course site on UM Learn (click on "Discussions" under "Communications"). Post questions and comments related to Assignment 1 there, and I will respond. Please do not post solutions, not even snippets of solutions there, or anywhere else. I strongly suggest that you read the assignment discussion group for helpful information.

Computer science help centre: The staff in the help centre can help you (but not give you assignment solutions!). See their website at http://www.cs.umanitoba.ca/undergraduate/computer-science-help-centre.php for location and hours. You can also email them at helpctr@cs.umanitoba.ca.

Programming Standards

When writing code for this course, follow the programming standards, available on this course's website on UMLearn. Failure to do so will result in the loss of marks.

Question

Remember: You will need to read this assignment many times to understand all the details of the program you need to write.

Goal: The purpose of this assignment is to write a Java program that times the execution of six sorting algorithms (variations on insertion sort, merge sort and quick sort) and reports on the run times and whether the algorithms actually sorted the list (using a testing method you will also write).

Code you can use: You are permitted to use and modify the code I gave you in class for the various sorting algorithms, but you are NOT permitted to use code from any other source. Any other code you need for this assignment, you must write for yourself. I encourage you to write ALL the code for this assignment yourself, based on your understanding of the algorithms — you will learn the material much better if you write the code yourself.

What you should implement: Implement the following seven algorithms (you can add any necessary private helper methods):

1. **A (non-recursive) insertion sort method:** Use the following header:

```
private static void insertionSort ( int[] a, int start, int end )
```

Take the insertion sort algorithm from class and modify it so that it sorts only positions `a[start]` to (and including) `a[end-1]` in array `a`, not touching any other position in the array.

Also, write a `public` driver method with the following header:

```
public static void insertionSort ( int[] a )
```

Its task is to simply call the above `private` method, passing the correct values to the `private` method's parameters so that it sorts the entire array.

2. **A recursive merge sort that does exactly what we talked about in class:** Take the algorithm exactly as discussed in class — the `public` driver `mergeSort` method (simply calls the private recursive method with the extra parameters it needs), the `private` recursive helper `mergeSort` method (does the merge sort), and the `merge` iterative helper method (merges two sorted sublists into one sorted list).

3. **An iterative merge sort algorithm:** Use the following header:

```
public static void iterativeMergeSort (int[] a)
```

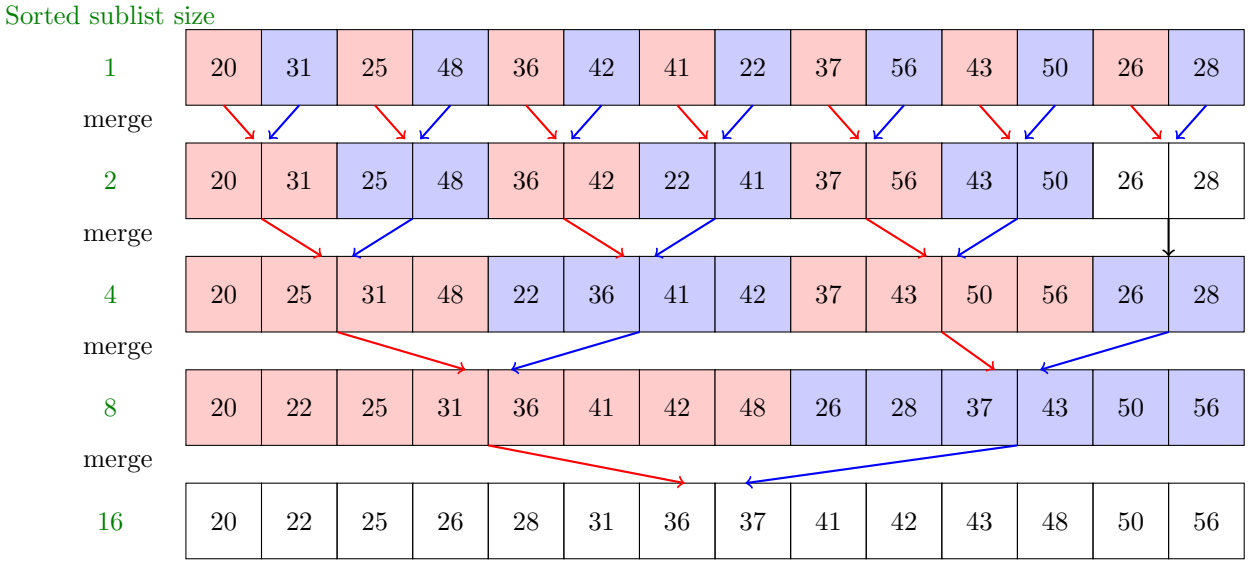
This method sorts the whole array `a`. It does a merge sort "from the bottom up" using loops and no recursion:

- It merges adjacent pairs of lists of size 1 (i.e., adjacent pairs of items) into sorted lists of size 2.
- Then it merges adjacent pairs of sorted lists of size 2 into sorted lists of size 4.
- Then it merges adjacent pairs of sorted lists of size 4 into sorted lists of size 8.
- And so on until the last step merges two sorted lists into one sorted whole.

Example: Consider the following 14-item list:

$$20, 31, 25, 48, 36, 42, 41, 22, 37, 56, 43, 50, 26, 28$$

The iterative method starts with each item as a (sorted) list of size one and begins merging:



Each round of merging doubles the size of the sublists. Notice the following:

- Look at any row of the above diagram. The last sublist in that row may not be as large as the others. That's because the number of items to be sorted might not be an exact multiple of the sublist size in that row. (E.g., look at the row with sublists of size 4: the last sublist in that row contains fewer than 4 items because the total list size is not a multiple of 4. Similarly, look at the next row with sublists of size 8 in the above example: the last sublist in that row contains fewer than 8 items because the total list size is not a multiple of 8.)

  (Fortunately, the `merge` method we talked about in class doesn't care whether the two lists it is merging are the same size or not.)
- Sometimes the last sublist in a row does not participate in the current round of merging. That's because there may be an odd number of sublists of the current size. (E.g., look at the row with sublists of size 2. There's an odd number of sublists of size 2 in that row, so the last sublist in the row doesn't participate in the merging in that round.)

Implementation notes: This method is a nested pair of loops:

- The outer loop is a `for`-loop that counts through the sublist sizes — it changes the loop index (the size) by multiplying it by 2. It stops when the sublist size is the number of items in the whole list or larger.
- The inner loop is a loop that moves through the pairs of sublists, merging adjacent pairs.

  Make sure that it copes with the last sublist properly.

Furthermore, the method creates only ONE array ONCE, a `temp` array for the `merge` step. It does NOT create new arrays for the sublists. (Creating arrays is expensive — don't create arrays that you don't have to.) Instead, it simply computes start and end indices in the original array for the sublists.

4. **An ordinary recursive quick sort algorithm:** Use the following header:

   ```
   private static void quickSort (int[] a, int start, int end )
   ```

   This method uses the quick sort algorithm discussed in class to sort `a[start]` to `a[end-1]`. It must choose its pivot using the median-of-three technique, as discussed in class.

   Note that the base cases for the quick sort using median-of-three pivot-choosing are lists of size 0 and 1 (do nothing) and lists of size 2 (swap the two items if the first item is larger than the second item).

   Also, write a `public` driver method with the following header:

   ```
   public static void quickSort ( int[] a )
   ```

   Its task is to simply call the above `private quickSort` method, passing the correct values to the `private` method's parameters so that it sorts the entire array.

5. **A hybrid recursive quick sort algorithm that uses a breakpoint:** Use the following header:

   ```
   private static void hybridQuickSort( int[] a, int start, int end, int breakpoint )
   ```

   This method is similar to the recursive `quickSort` algorithm above, except it has an extra base case:

   - If `a[start]` to (and including) `a[end-1]` is 0 or 1 item, do nothing (same as before).
   - If `a[start]` to (and including) `a[end-1]` is 2 items, swap them if the first one is larger than the second (same as before).

- If `a[start]` to (and including) `a[end-1]` consists of more than two items, but no more than `breakpoint` items, then call the `private insertionSort` method above to sort `a[start]` to (and including) `a[end-1]` (this case is a new base case).
- If `a[start]` to (and including) `a[end-1]` consists of more than `breakpoint` items, then do the usual quick sort steps (choose a pivot using the median-of-three technique, partition the items using the chosen pivot, and finally recursively call `hybridQuckSort` twice to sort each of the smalls and the bigs.

(Make sure its recursive calls are to `hybridQuickSort`, NOT to `quickSort`.)

Also, write a `public` driver method with the following header:

```
public static void hybridQuickSort ( int[] a, int breakpoint )
```

Its task is to simply call the above `private hybridQuickSort` method, passing the correct values to the `private` method's parameters so that it sorts the entire array.

6. **A mutually-recursive sorting algorithm:** This algorithm alternates between quick sort and merge sort using mutual recursion. For this sort, you must write three methods.

   The first method uses the following header:

   ```
   private static void mutualQuickSort( int[] a, int start, int end, int[] temp )
   ```

   It has the same body as an ordinary recursive quick sort, except that instead of calling itself to sort the smalls and again to sort the bigs after partitioning (in the recursive part), it calls `mutualMergeSort` (see the description immediately below) to sort the smalls and again to sort the bigs.

   The second method uses the following header:

   ```
   private static void mutualMergeSort( int[] a, int start, int end, int[] temp )
   ```

   It has the same body as an ordinary recursive merge sort, except that instead of calling itself to sort the two halves after computing the midpoint (in the recursive part), it calls `mutualQuickSort` (see the description immediately above) to sort the two halves.

   The third method is the public driver method that the user calls. It has the following header:

   ```
   public static void mutuallyRecursiveSort( int[] a )
   ```

   It should create a `temp` array (just like the public driver method for ordinary recursive merge sort), and then call the `mutualQuickSort` method (above), passing it the appropriate parameters so that the whole array is sorted.

7. **An error-checking method:** Use the following header:

   ```
   private static boolean isSorted( int[] a, int start, int end )
   ```

   This method makes sure that `a[start]` to `a[end-1]` is sorted. It should return a boolean: it should return `true` if `a[start]` to `a[end-1]` is sorted (that is, each item (after the first item) is at least as large as the previous item) and `false` otherwise.

   Also, write a `public` driver method with the following header:

   ```
   public static boolean isSorted( int[] a )
   ```

Its task is to simply call the above `private isSorted` method, passing the correct values to the `private` method's parameters so that it checks the entire array, and returning whatever the `private` method returns.

Main method: Finally, the `main` method should call helper methods to perform the following tasks: The first task is to read in the numbers to be sorted from an array. Prompt the user for a file name and then read integers from the file into an array. To make file input simple, the first integer (on the first line of the file) tells you how many integers are in the rest of the file (i.e., not including the first integer). Thus, the first integer tells you how big to make the input array and you can use a simple `for`-loop to read in the rest of the file into the array.

The second task is to find the average running time of each of the above sorting methods on the input array. (Pass breakpoint 307 to the hybrid quick sort.) Time each sorting method 50 times and compute the average of the 50 timings.

For each call to a sorting method, you should perform the following steps:

- Copy the input array into a second array. (You have to copy the input array for each sorting call and pass the copy to the sorting method, or else you will be resorting a sorted array!).
- Sort the new copy of the array using the sorting method, timing how long the method took (using nanosecond precision). You can time a method as follows:

```
import java.util.*;
...
  long startTime, endTime, elapsedTime;
...
  startTime = System.nanoTime();
  // call your method here
  endTime = System.nanoTime();
  elapsedTime = endTime - startTime;
```

  (`System.nanoTime()` gives nanosecond precision, but not necessarily nanosecond accuracy.)

- Run your error-checking method on the sorted array after each trial (when you have called and timed a method), and print out an appropriate error message only if the array is not sorted.

Your output should include the average running time of each sorting method over 50 trials, the size of the array sorted and the appropriate error messages if the error-checking method found any errors in the output from each method, all with appropriate headers.

Furthermore, your output should list which sorting method had the fastest average running time, so your `main` method must keep track of the time and name of the sorting method with the minimum average time. Thus, your output will look something like the following:

```
Comp 2140  Assignment 1   Fall 2016
Sorting with insertion sort, merge sort, and quick sort.
Model Solution

Enter the input file name (.txt files only):
sortingInput5.txt

*********************************************

Average time for insertion sort: 176762720 nanoseconds.
Average time for recursive merge sort: 5507020 nanoseconds.
Average time for iterative merge sort: 3732220 nanoseconds.
```

```
Average time for quick sort: 3960480 nanoseconds.
Average time for hybrid quick sort: 3220920 nanoseconds.
Average time for mutually recursive merge sort and quick sort: 11096640 nanoseconds.

*************************************************************************

The best sorting method is hybrid quick sort.
It sorted all 30000 numbers in an average time of 3220920 nanoseconds over 50 trials.

*************************************************************************


Program ends.
```

I will provide five input files for you to work with — they are available with this assignment in the content browser of the course website on UM Learn.

Suggestions

To easily prompt the user for a file name, you can use use keyboard input. A few lines like the following will probably do the right job:

```
Scanner keyboard;
String fileName;
Scanner file;

// Allow user to choose file with keyboard input.
keyboard = new Scanner( System.in );
System.out.println( "
nEnter the input file name (.txt files only): " );
fileName = keyboard.nextLine();

try
{
  file = new Scanner( new File( fileName ) );


}
catch (IOException e)
{
  System.out.println("IO Error: " + e.getMessage());
}
```

(The input file needs to be in the same directory as your program.)

Use `System.arraycopy( )` to copy arrays. A call like

```
System.arraycopy(x, m, y, n, len);
```

will copy `len` positions from array `x` starting at position `m` into array `y` starting at position `n`; that is, `x[m]`, `x[m+1]`, `x[m+2]`, ... `x[m+len-1]` are copied into `y[n]`, `y[n+1]`, `y[n+2]`, ... `y[n+len-1]`.

The timing of a method can be affected by other things that your computer is doing while you are testing. So shut down other programs, and turn off the internet, to minimize interference with the timings. The Java garbage collector can interfere with timing, but there is not much you can do about that.

Hand-in Instructions

Go to COMP2140 in UM Learn, then click "Dropbox" under "Assessments" at the top. You will find a dropbox folder called "Assignment 1". (If you cannot see the assignment dropbox, follow the directions under "Honesty Declaration" below.) Click the link and follow the instructions. Please note the following:

- Submit ONE .java file only. The .java file must contain all the source code. The .java file must be named
  `A1<your last name><your student id>.java` (e.g., `A1Cameron1234567.java`).
- Please do not submit anything else.
- You can submit as many times as you like, but only the most recent submission is kept.
- We only accept homework submissions via UM Learn. Please DO NOT try to email your homework to the instructor or TA or markers — it will not be accepted.
- We reserve the right to refuse to grade the homework or to deduct marks if you do not follow instructions.
- **Assignments become late immediately after the posted due date and time.** Late assignments will be accepted up to 49 hours after that time, at a penalty of 2% per hour or portion thereof. After 49 hours, an assignment is worth 0 marks and will no longer be accepted.

Honesty Declaration

To be able to hand in any assignment (i.e., to make the dropbox visible to you), you must first agree to the blanket honesty declaration. This declaration covers ALL your work in the course. To agree to the honesty declaration:

- Go to the COMP 2140 homepage on UM Learn, look at the main header (the navigation bar) at the top. Along with "Resources", "Communication", and "Assessments", you will see "Checklist" — click on "Checklist".
- If you can see a blue "Honesty Declaration" above the red "Note" near the top (and you canNOT see the "Save" option at the bottom), click on the blue "Honesty Declaration" at the top.
- Read the honesty declaration, check off "Agreement" (verifying that you have read the honest declaration and agree to it), and finally, click "Save". Only after this task is completed will you be able to see any dropboxes.

You only have to agree to the honesty declaration once, because it covers all your work in the course.