

LABORATORY **12**

Sorting Out Sorts

A binary search depends on having a sorted array. If a large number of searches are to be done, then it is efficient to sort the array and use the binary search. If only one or two lookups are to be done, then a linear search is more efficient as we omit the time required to sort the array.

How much time does it take to sort an array of size N ? The answer depends on the sorting algorithm used. In this laboratory we will examine a new sort called the *merge sort* and compare it with the *selection sort* that was introduced in an earlier laboratory. Our comparison will be based on timings of the algorithms coded in Java. We will explore Java's mechanism for keeping time and discuss problems in collecting timing data.

The post-laboratory section will introduce a third algorithm called *quick sort* and discuss its implementation and efficiency.

Pre-Laboratory Reading

The Merge Sort Algorithm

From the laboratory on recursive methods, we recall the basic description of the merge sort algorithm. Assume that if you have two lists that are *individually sorted*, you know how to merge them into a single sorted list. You examine the initial items on each list and move the smaller one to a new list, then advance to the next item of the list where the smaller value was found. You continue to do this until one list is empty and then just move all of the items on the non-empty list to the new combined list. This operation is called MERGE.

The *merge sort* works as follows when given a list:

- If the list has one item or no items, quit/return.
- Split the list into two halves and apply merge sort to each half.
- Apply MERGE to the two sorted halves.

Let us now be more formal in the statement of the algorithm; in particular, we need to be more explicit in describing the MERGE algorithm.

MERGE

Let A and B be sorted lists. Assume that the first and last positions of A are `first_A` and `last_A`. Assume that the first and last positions of B are `first_B` and `last_B`. Assume that `first_A <= last_A` and `first_B <= last_B`.

Let M be an array of size large enough to hold all of the elements of A and B that are to be merged. Its initial index will be `first_M`. The array M will be the merger of the other two arrays.

Use three integer variables `a`, `b`, and `m` refer to positions in A, B, and M, respectively. Their initial values are `first_A`, `first_B`, and `first_M`, respectively.

Repeat the following step until either `a > last_A` or `b > last_B`, which indicates that the list A or the list B, respectively, is emptied.

- If `A[a]` comes before `B[b]`, then set `M[m]` to `A[a]` and increase both `m` and `a` by 1. Otherwise set `M[m]` to `B[b]` and increase both `m` and `b` by 1.

To complete the algorithm, determine which list is not yet empty and copy all of its remaining elements to the end of M.

When MERGE is coded as a procedure, it will have eight parameters: A, `first_A`, `last_A`, B, `first_B`, `last_B`, M, and `first_M`. The array M will have any existing values overwritten during the algorithm, while A and B will have no changes in their elements.

Merge Sort

Once MERGE is written we can describe merge sort precisely.

Assume we have an array T with initial position `first_T` and final position `last_T`. We wish to sort `T[first_T]`, . . . , `T[last_T]`.

- If `last_T <= first_T`, quit/return as there is at most one item.
- Let `middle` be $(\text{first_T} + \text{last_T}) / 2$, calculated using integer division.
- Apply merge sort to T from positions `first_T` to `middle`.
- Apply merge sort to T from positions `middle + 1` to `last_T`.
- Create an array `temp` that has `last_T - first_T + 1` cells.
- Apply MERGE to the two (now sorted) halves of T using `temp` as the array where the merged halves are placed.

- Copy the contents of `temp` to `T` from `first_T` to `last_T`. In `temp` the initial position is 0.

The last two steps must be separate. We cannot merge the two halves of `T` directly into `T`, as we would overwrite some values. This is why we use an array `M` in the algorithm for `MERGE`. The array `temp` above would be passed as a parameter that corresponds to `M` when the `MERGE` procedure is called.

Efficiency

The efficiency of merge sort is not obvious because of the recursion. The total number of comparisons in `MERGE` is at most the sum of the number of items in the arrays `A` and `B`. This is because an item gets moved to `M` after each comparison.

If the number of items in `T` is a power of 2, then it is not hard to show via algebra that the number of comparisons and copies is proportional to $N \cdot \log_2(N)$ where N is the number of items in `T`. (We omit the algebra and save the analysis for a later course.) Compare this with the time proportional to $N \cdot N = N^2$ for the selection sort.

Generating Random Data for Tests

We will want to test the coding of sort algorithms on large sets of random data. We also will want to compare the time it takes to sort large arrays of random values using each coded algorithm.

We can generate such arrays using Java's `Random` class, which is in the package `java.util`. The code below shows how this might be done using the `nextDouble()` method, which returns a pseudorandom¹ value between 0.0 and 1.0.

```
double[] d = new double[4096];
Random r = new Random();
for (int i = 0; i < 4096; i++)
    d[i] = r.nextDouble();
```

The code will fill the array `d` with 4,096 random values, each in the range 0.0 to 1.0.

Timing

Java has a way to represent the current time by creating a `Date` object with the information about the current moment. `Date` is one of the classes in the `java.util` package. The `Date` object is initialized with the number of milliseconds since January 1, 1970. This is a very large integer and is stored as a `long` type value. The method `getTime()` of the `Date` class returns this `long` value. The code below shows how a simple timing might be done. Notice that we create a `Date` object and then immediately call its `getTime` method—we don't even name the `Date` object.

```
// DateTimeTest.java
// just tests Date object's use for measuring time

import LabPkg.*;
```

-
1. The term "pseudorandom" is used because such values are not truly random in the mathematical sense. Each value is generated from the previous value using an algorithm that attempts to make any value between 0.0 and 1.0 have the same probability of being generated. The first value's calculation is based on a user-provided value called a "seed" or is based on the current time.

```

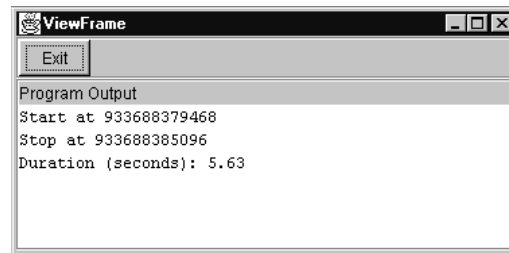
import java.util.*;

public class DateTimeTest
{
    public static void main(String[] args)
    {
        ViewFrame vf = new ViewFrame();
        vf.setVisible(true);

        long timeStart = new Date().getTime(); // initial time
        Useful.pause(56); // event being timed
        long timeStop = new Date().getTime(); // stopping time
        vf.println("Start at " + timeStart);
        vf.println("Stop at " + timeStop);
        vf.println("Duration (seconds): "
+ Toolbox.roundDouble((timeStop - timeStart) / 1000.0,2));
    }
}

```

We really only care about *elapsed time*, which is the final value printed. We divide by 1000.0 to convert the milliseconds to seconds as a double, but we round the result to two decimal places using one of our Toolbox utility methods. The `Useful.pause(56)` should cause a delay of 5.6 seconds. The result of a run is shown below.



Notice that the result is not 5.60. This is due to the small amount of time it takes to do the creation of the second `Date` object. If this program is run several times there will be some small differences in that second decimal place due to system overhead during object creation and the crudeness of Java's time-keeping. If the event being timed is of short duration, then the inaccuracy in the measurement can be quite large as a fraction of the elapsed time.

Review Questions

1. What method of the `Random` class is used to get a random value between 0.0 and 1.0?
2. What Java class can be used to get information about the current time?
3. If sorted arrays `X` and `Y` have 20 and 15 items respectively, what is the maximum number of comparisons done by a `MERGE` of `X` and `Y`?
4. If an array `A` has 1,024 items, the time to sort it using merge sort is proportional to what value?

The Laboratory

A Guide to the Laboratory

- Tasks 1 and 2 demonstrate the generation of random data for testing and the vagaries of timing using Java's `Date` class.

- Tasks 3 and 4 require you to write and test the methods `MERGE` and `mergeSort`. The methods are placed in your `Toolbox` class.
- Task 5 uses a provided program to collect timing data for sorting large arrays using selection sort and merge sort. You are asked to draw conclusions from the analysis of the data.

Task 1: Generating Random Data

Write a simple program called `RandomGeneration.java` that will request `N`, the number of values desired, and then generate a list of `N` random double values. Use a `ViewFrame` for input and output.

Turn in a printout of the program.

Task 2: Timing Variability

Download the file `DateTimeTest.java` and modify it to ask the user to enter the pause amount (rather than use 56). Be sure that the code asking for the pause amount is not part of what is being timed! You can solve this problem by putting the code to request the pause amount immediately before the declaration of the variable `timeStart`.

- Run the program four times with the pause value of 10 (1 second) and record the average duration time.
- Run the program four times with the pause value of 100 (10 seconds) and record the average duration time.
- Run the program four times with the pause value of 200 (20 seconds) and record the average duration time.

What conclusions do the data suggest concerning the accuracy of timing using this method? Write this up to turn in along with the data you collected.

Task 3: Writing and Testing `MERGE`

For each of the methods `MERGE` and `mergeSort` that you are to write, you should do versions for integer and double arrays. We will not need a `String` version for this laboratory.

Add the method below to your `Toolbox` class following the algorithm in the pre-laboratory reading.

```
public static void MERGE(int[] A, int first_A, int last_A,
                        int[] B, int first_B, int last_B,
                        int[] M, first_M)
```

Be sure to add the comments.

Write a test program that will create two sorted arrays and call `MERGE`. Display the resulting merged array in a `ViewFrame` output window. The two sorted arrays might be “manually” created via a declaration like

```
int[] T = {20, 30, 34, 35, 56, 70};
int[] S = {1, 2, 3, 25, 36};
```

The array to hold the merged values of `T` and `S` would be declared as

```
int[] Ans = new int[T.length + S.length];
```

Don’t forget to do a version where the parameters `A` and `B` are arrays of type `double` and the method fills an array of type `double`. It is also useful to have a version for arrays of `String` objects but that is optional at this time.

Task 4: Coding and Testing Merge Sort

Using the algorithm in the pre-laboratory reading, add the following methods to your Toolbox class:

```
public static void mergeSort(int[] A, int first, int last)
public static void mergeSort(double[] A, int first, int last)
```

You can write a simple test program to create an array of random data values, call `mergeSort`, and then use `isSorted` to see if it worked properly. The input might be the number of items to be sorted and the output could be a simple message indicating “Success” or a “Problem.”

Create a printout of your test program to turn in. Turn in a printout of the pages of `Toolbox.java` containing the `MERGE` and `mergeSort` methods. Be sure your name is on the pages.

Task 5: Finding the Better Sort

We want to make three claims and see if we can collect empirical evidence to support them.

- **Claim #1:** For reasonably large values of N , the array size, the time it takes selection sort to sort the entire array is proportional to N^2 for random data.
- **Claim #2:** For reasonably large values of N , the array size, the time it takes merge sort to sort the entire array is proportional to $N \log_2(N)$ for random data.
- **Claim #3:** Merge sort is significantly faster than selection sort, and the improvement increases with N .

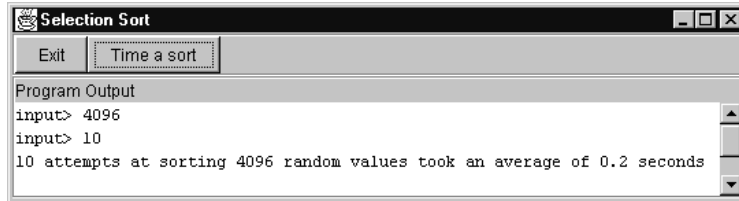
We now want to run selection sort and merge sort on large random data sets and record their running time. The sizes of the data sets must be at least 4,000, because for smaller size arrays the time for the sorts cannot be measured precisely enough to reliably detect the real time differences. Since we claimed that the time for merge sort is proportional to $N \log_2(N)$, it is useful to use powers of 2 for the values of N .

The following values should be used for array sizes in the tests:

```
4096
8192
16384
32768
65536
```

You will create a project containing two files—`SortTester.java` and `SortTest.java` (both files are available for downloading)—plus your `Toolbox.java` file.

The file `SortTest.java` will have a short `main` method that will create two instances of `SortTester`, one that will use the selection sort and one that will use the merge sort. Each will have an action button labeled “Time a sort” that will pop up a request for the array size to be used, then request how many arrays to sort, and finally create the arrays of that size and fill them with random values. Then it will time how long the sort operations take for all of those arrays. The output will show the average time for each of the arrays to be sorted. The `SortTester` object will have a `ViewFrame` object, and the title of the `ViewFrame` object will be the name of the sort being tested. The output area of the `ViewFrame` object will contain the results of the test. Be sure that `setIOEcho(true)` is called so you can be sure you record the array size properly. Examples of the two `SortTester` objects are shown below.



You should have the following static data values of the class `SortTester`, which are used with the constructor to signal which sort is to be used.

```
public static final int SELECTIONSORT = 1;
public static final int MERGESORT = 2;
public static final int QUICKSORT = 3;
```

The last one will be used when we test another useful sort later.

Once you have compiled the project, run it and time each sort using the sizes specified in the table below and record the results. The program displays the time in seconds to at most three decimal places. The program allows you to specify how many sortings will be timed (up to 10 in any single run) and produces the average sorting time. This is designed to reduce the role of overhead in creating the timing objects.

N	Log ₂ (N)	Selection Sort Time	Merge Sort Time
4096	12	(5)	(5)
8192	13	(4)	(5)
16384	14	(3)	(5)
32768	14	(2)	(5)
65536	15	(1)	(5)

The number of sorts per run is indicated in parentheses in the table. Fill in the average time per run. **Warning:** You will find that the selection sort may take a *very* long time to do the 65536 size array sort.

Please note that because Java does not measure time as accurately as desired and because the time can be affected by other programs running on the computer you will get “noise” in your data. Do the best you can to answer the questions below.

1. What conclusions can you draw from this comparison?
2. Which claims are supported?
3. Does it appear that the selection sort’s time is proportional to N^2 ?
4. Does it appear that the merge sort’s time is proportional to $N \cdot \log_2(N)$?

Write up your data and your conclusions for submission. Be sure to use proper grammar. Be neat.

Post-Laboratory Reading

Quick Sort

Quick sort, or quicksort, is another recursive sort that has a running time proportional to $N * \log_2(N)$ where N is the number of items. For random data it is faster than merge sort because it has a smaller constant of proportionality. However, if the data is already nearly in order, the usual algorithm can be very slow, like the selection sort.

There are clever ways to code quick sort and you will explore them in another course or you can find details in your textbook. We simply give the algorithm, where as usual A will be an array to be sorted from position `first` to position `last`.

- If `last <= first`, then quit/return as there is at most one item.
- Select one element of the array and call it the “pivot.”
- Shuffle the elements of the array so that all the elements that are less than or equal to the pivot element (including the pivot element itself) are at the beginning of the array and all the elements greater than the pivot are at the end of the array. Let P be the position where the elements larger than the pivot start. Hence, all the elements $A[\text{first}], \dots, A[P - 1]$ are less than or equal to the pivot and all elements $A[P], \dots, A[\text{last}]$ are larger than the pivot. By doing a swap, make sure that the original pivot value is in $A[P - 1]$.
- Call quick sort on A from `first` to $P - 2$.
- Call quick sort on A from P to `last`.

The “shuffle” step guarantees that the selected pivot element is put in its proper place in the sorted array, i.e., the place where it will end up when the whole array is sorted. One coding approach that works is to create a temporary array of the appropriate size, and copy items from A to it so that the shuffle step is accomplished. Then copy the shuffled items back to A . (The best implementation of the algorithm is able to do the shuffle step without a temporary array.)

If the data is random and the pivot is a random element, then the position P will be approximately in the middle from `first` to `last` with high probability. Hence the recursive calls are dealing with arrays that are only half as large. Of course, this is a probabilistic statement, and it is rare that an array splits exactly in half.

Quick sort gets into trouble and is slow if the pivot element is the largest or smallest item of the array. Then one recursive call has an empty array while the other has the entire array except for the pivot element. Hence, it looks like the recursive version of selection sort in terms of time.

Analysis of quick sort is difficult because of the dependence on the value of the pivot. Experimentation can convince you that it is superior to merge sort for random data.

Exercise

Add a method `quickSort` to `Toolbox.java` following the algorithm above. Test it for correctness using random data and the `isSorted` method. Then use `SortTester.java` and a modified `SortTest.java` to collect timing data. Is it faster than merge sort on random data?