Working With Big Data

Lesson Objectives

When you finish this lesson you will be able to:

- characterize the storage requirement for an algorithm.
- access the contents of a structured binary file in the same way that you would an array stored in main memory.
- read from and write to a memory-mapped file.

Working with Big Data

What if you had to sort a collection of integers? The following example shows how to use the built-in sorting capabilities provided by the JDK.

Create a **BigData** project, and assign it to the **Java6_Lessons** working set.

Then, create a **SortRandomIntegers** class in the default package of the **/src** source folder:

CODE TO TYPE: SortingExample

```
import java.util.Arrays;

public class SortRandomIntegers {
  public static void main(String[] args) {
    int numIntegers = 1000;
    int[] group = new int[numIntegers];

    for (int i = 0; i < numIntegers; i++) {
      group[i] = (int)(Math.random()*numIntegers);
    }

    Arrays.sort(group);

    for (int i = 0; i < 10; i++) {
      System.out.println(group[i]);
    }
  }
}
```

Run the code to verify that it prints out ten numbers in sorted order.

This small program generates a random array containing 1000 integers, sorts them, and prints out the smallest ten in the array. You should always use the **Arrays.sort** built-in methods to sort arrays because it provides tuned algorithms with a performance that is nearly always *O(n log n)*. For *comparison-based* sorting algorithms (where you can only sort the elements by directly comparing the individual elements) this is the best we've got.

Now what if you have a large collection of integers? Like 450 million? If you modify the settings of the above program to generate **450000000** integers instead of **1000** integers, you'll see this error message:

OBSERVE: Unable to create large arrays in memory

```
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
        at SortRandomIntegers.main(SortRandomIntegers.java:8)
```

The primary issue with this code is that it is simply impossible to create a contiguous array to contain a large collection of elements. You can try to increase the heap space available to your Java virtual machine, but eventually the computer on which you are running will exhaust its available memory. So how is it possible to deal with extremely large data sets? You will need to develop techniques that manage the transfer of data from external storage (such as a hard disk) into main memory (what is commonly called RAM). In the early days of computing, main memory was measured in kilobytes (not gigagbytes!) and programmers learned how to work within these constraints. In this era of "Big Data" where data can be measured in terabytes and petabytes, even modern programmers have to make some fundamental adjustments.

In this lesson, you'll learn how to sort large sets stored on disk, sets that may be too large to store in main memory. We'll show examples using small data sets, but they can scale to much larger data sets as needed.

Sorting Large Sets Using External Storage

Most sorting algorithms operate over an array of values, swapping elements in the array until the elements are in order. The earlier **sort** method does that. However, when the number of elements being sorted is too large to store in main memory, there are sorting algorithms that allow us to use external storage. The fundamental algorithm to learn is called *MergeSort*. You've probably used this technique in the real world already. Suppose that you had a stack of 50 notecards, each containing a single number. To sort the whole stack, divide it into two stacks of 25 notecards each. Sort each of these two stacks individually, which results in two sorted stacks of notecards where you can see the topmost visible card in each stack. You can "merge" these two smaller stacks into a third sorted stack by repeatedly taking the card whose visible number is the smaller of the two. This merging process gives the algorithm its name.

MergeSort is recursive, since it breaks up a problem instance into two smaller instances of half the size. To stop the recursion, consider two cases:

1. Sorting a collection of two values: swap the first value with the second if they are out of order.
2. Sorting a collection with a single value: the collection is already sorted, so stop.

You have enough information to write the pseudocode now. The notation |A| represents the size of the collection A:

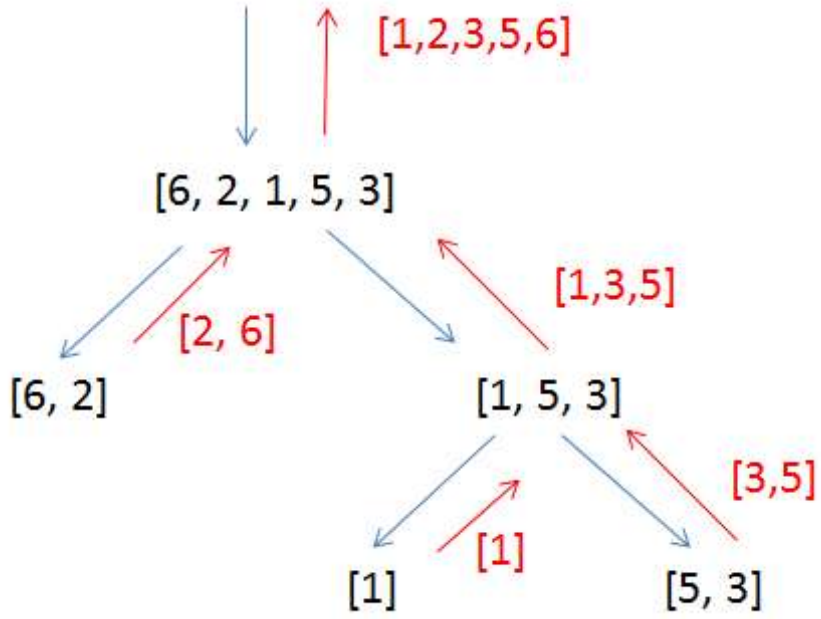
OBSERVE: pseudocode for Mergesort

```
MergeSort (A)
  if |A| < 2 then return A
  if |A| = 2 then
    swap elements of A if out of order
    return A

  sub1 = MergeSort(left half of A)
  sub2 = MergeSort(right half of A)

  merge sub1 and sub2 into a new array B
  return B
```

Try this out by manually executing MergeSort on the collection [6, 2, 1, 5, 3]. In the following graphic, the blue arrows represent invocations of *MergeSort* and the red arrows represent the returned sorted arrays. The newly created arrays are depicted in red and contain three or more elements (that is, [1,3,5] and [1,2,3,5,6]):



Now it's your turn to implement this algorithm:

Create a **sort** package in the **/src** source folder.

Create a **CopyMergeSort** class in the **sort** package as shown.

CODE TO TYPE: CopyMergeSort class

```
package sort;

import java.util.Arrays;

public class CopyMergeSort {
  public static void main(String[] args) {
    int[] group = new int []{6, 2, 1, 5, 3};
    group = copymergesort(group);
    for (int i : group) {
      System.out.print (i + " ");
    }
  }

  static int[] copymergesort(int[] A) {
    if (A.length < 2) {
      return A;
    }

    if (A.length == 2) {
      if (A[0] > A[1]) {
        int tmp = A[0];
        A[0] = A[1];
        A[1] = tmp;
      }
      return A;
    }

    int mid = A.length/2;
    int[] left = Arrays.copyOfRange(A, 0, mid);
    int[] right = Arrays.copyOfRange(A, mid, A.length);

    left = copymergesort(left);
    right = copymergesort(right);

    for (int i = 0, j = 0, idx=0; idx < A.length; idx++) {
      if (j >= right.length || (i < left.length && left[i] < right[j])) {
        A[idx] = left[i++];
      } else {
        A[idx] = right[j++];
      }
    }

    return A;
  }
}
```

Run this code to verify that it works. You might experiment with different initial arrays to see how the code handles arrays with just 1 or 2 (or larger number of) elements:

OBSERVE: Output of CopyMergeSort

```
1 2 3 5 6
```

Let's take a closer look at this code. The base cases of the recursion are as follows:

OBSERVE: Base cases of CopyMergeSort recursion

```
  static int[] copymergesort(int[] A) {
    if (A.length < 2) {
      return A;
    }

    if (A.length == 2) {
      if (A[0] > A[1]) {
        int tmp = A[0];
        A[0] = A[1];
        A[1] = tmp;
      }
      return A;
    }

    ...
}
```

The **copymergesort** method must return an **int[]** array, so these **if** statements both return the input array, **A**. When there are two elements in the array, the second **if** statement **swaps the two elements** if they are out of order.

OBSERVE: Recursive steps

```
   int mid = A.length/2;
   int[] left = Arrays.copyOfRange(A, 0, mid);
   int[] right = Arrays.copyOfRange(A, mid, A.length);

   left = copymergesort(left);
   right = copymergesort(right);
```

The true logic of this algorithm occurs when the array **A is subdivided into two arrays**, **left** and **right**, which are then recursively sorted using **copymergesort**.

OBSERVE: Merging two sorted arrays

```
  for (int i = 0, j = 0, idx=0; idx < A.length; idx++) {
    if (j >= right.length || (i < left.length && left[i] < right[j])) {
      A[idx] = left[i++];
    } else {
      A[idx] = right[j++];
    }
  }

  return A;
}
```

Once the two recursive calls return, **left** and **right** will be sorted (this is the fundamental property of any recursive function). All that remains is the process of selecting the smaller of the two elements while merging these two lists. The code above reuses array **A** to store the sorted values. Variable **i** will iterate over the indices in **left**, while **j** will iterate over the indices in **right**. **idx** identifies the index location in **A** into which the smaller value of **left[i]** or **right[j]** will be written. The loop terminates once all values have been transferred into **A** (that is, when **idx** = **A**.length). Once **right** has exhausted its elements (because **j >= right.length**), elements of **left** are transferred to **A**. Similarly, once **left** has exhausted its elements (because **i >= left.length**), elements of **right** are transferred to **A**.

This code works, and it's reasonably efficient on small sets of numbers, but we also need space for the **left** and **right** arrays. To address this issue, you need to learn how to characterize the storage requirements for an algorithm.

Characterizing Storage Requirements for an Algorithm

Throughout this course, you have characterized the running time of an algorithm to determine its efficiency. This is how algorithms are most often compared. You can also compare algorithms by their storage requirements. There is a "Time vs. Space" tradeoff in programming that explains many of the design decisions that a programmer must make. For example, each Java class that is used as a key value in a **HashMap** must implement a **hashCode()** method as part of the Collections Framework. As we've mentioned, if two objects are equal to each other, then the value returned by **hashCode** must also be the same. For immutable classes (such as **String**), a program can save

computation time by computing the hash value just once and then caching the result for subsequent invocations. Here is the code from **java.lang.String**:

OBSERVE: String.hashCode() method

```
public int hashCode() {
  int h = hash;
  int len = count;

  if (h == 0 && len > 0) {
    int off = offset;
    char val[] = value;

    for (int i = 0; i < len; i++) {
       h = 31*h + val[off++];
    }
    hash = h;
   }

  return h;
}
```

Whenever **hashCode** is executed, it checks to see if the cached value **hash** is equal to zero; only then does it compute and store the value in the **hash** class attribute. This code is more efficient because of the extra integer being stored. How much extra storage is required? In this case it's a fixed amount of storage—just one additional **int** value. When addressing more complicated algorithms, you will need to determine whether the amount of extra storage is fixed, or is based on the size of the problem instance. For example, if you needed *2*n* additional stored array elements to sort an existing array of *n* elements, you would characterize the storage requirements as being *O(n)*. If, however, you needed *n*n* additional array elements to sort an existing array of *n* elements, the required storage is $O(n^2)$. We use the following notation in this course. *T(n)* refers to the *running time* characterization of an algorithm, *S(n)* refers to the *storage requirements* of an algorithm. Modify the **CopyMergeSort** code as shown:

CODE TO TYPE: Modifications to CopyMergeSort to compute storage requirements

```
package sort;
import java.util.Arrays;

public class CopyMergeSort {
  static int total=0;
  public static void main(String[] args) {
    int[] group = new int []{6, 2, 1, 5, 3};
    int numIntegers = 512;
    for (; numIntegers < 65536; numIntegers *= 2) {
      int[]group = new int[numIntegers];

      for (int i = 0; i < numIntegers; i++) {
        group[i] = (int)(Math.random()*numIntegers);
      }
      total = 0;
      group = copymergesort(group);
      System.out.println(total + " locations for " + numIntegers);
    }
    for (int i : group) {
      System.out.print (i + " ");
    }
  }

  static int[] copymergesort(int[] A) {
    if (A.length < 2) {
      return A;
    }
    if (A.length == 2) {
      if (A[0] > A[1]) {
        int tmp = A[0];
        A[0] = A[1];
        A[1] = tmp;
      }
      return A;
    }

    int mid = A.length/2;
    int[] left = Arrays.copyOfRange(A, 0, mid);
    int[] right = Arrays.copyOfRange(A, mid, A.length);

    left = copymergesort(left);
    right = copymergesort(right);

    for (int i = 0, j = 0, idx=0; idx < A.length; idx++) {
      if (j >= right.length || (i < left.length && left[i] < right[j])) {
        A[idx] = left[i++];
      } else {
        A[idx] = right[j++];
      }
    }

    total += A.length;
    return A;
  }
}
```

Execute this revised code to produce this table:

OBSERVE: Output showing storage requirements for CopyMergeSort

```
4096 locations for 512
9216 locations for 1024
20480 locations for 2048
45056 locations for 4096
98304 locations for 8192
212992 locations for 16384
458752 locations for 32768
```

When sorting 512 elements you need 8 times as much temporary storage; worse, when sorting 2,048 elements you need 10 times as much temporary storage. Based on the above table, when sorting *n* elements you need *2*n*log2(n)* temporary storage where *log2(n)* is the logarithm of *n* in base 2. So, the storage requirement for CopyMergeSort is *O(n log n)*. Even though CopyMergeSort executes efficiently, there is a serious issue regarding its storage requirements. Can something be done to remedy this? Yes.

MergeSort with O(n) Storage Requirements

Most sorting algorithms already perform "in place" with no additional storage requirements, so you might think that some intermediate compromise can be reached to reduce the storage requirements. You don't need to instantiate two sub-arrays **left** and **right** if you **instead pass parameters that refer to subranges within the array itself**. Let's start by revising the pseudocode for *MergeSort* to create a method that takes an array, **A**, and two internal indices, **[start, end)** where index location **start** is **inclusive** in the range **0 .. A.length-1** while **end** is **exclusive** in the range **0 .. A.length**. So, to sort an array one would invoke **MergeSort(A, 0, A.length)**. Note that the sorting is done "in place" so an array is no longer returned by this function.

OBSERVE: potential revised pseudocode for Mergesort

```
MergeSort (A, start, end)
  if end - start < 2 then return
  if end - start = 2 then
    swap elements of A if out of order

  mid = (end + start)/2;
  MergeSort(A, start, mid);
  MergeSort(A, mid, end);

  merge A's left- and right- sorted sub-arrays
```

The trouble with this approach is that merging in place will ultimately require just as many comparisons (and possibly more element swaps) as sorting in place. To avoid this situation, consider making these change to the pseudocode which introduces a copy of the initial array being sorted, which means the storage requirement is *O(n)*:

OBSERVE: final pseudocode for Mergesort

```
MergeSort (A)
  copy = copy of A
  MergeSort (copy, A, 0, |A|)

MergeSort (A, result, start, end)
  if end - start < 2 then return
  if end - start = 2 then
    swap elements of result if out of order

  mid = (end + start)/2;
  MergeSort(result, A, start, mid);
  MergeSort(result, A, mid, end);

  merge A's left- and right- sorted sub-arrays
  merge left- and right- of A into result
```

Because **copy** is a true copy of the entire array, the terminating base cases of the recursion will work *because they reference the original elements of the array directly at their respective index locations*. This observation is a sophisticated one; when you run this implementation in the debugger, you can validate it for yourself. In addition, the final merge step requires only *O(n)* operations.

Now it's your turn to implement this pseudocode.

In the **sort** package of the **/src** source folder, create a **MergeSortInteger** class as shown.

COE TO TYPE: MergeSortInteger class

```
package sort;

import java.util.Arrays;

public class MergeSortInteger {
  public static void main(String[] args) {
    int numIntegers = 1024;
    int[] group = new int[numIntegers];
    for (int i = 0; i < numIntegers; i++) {
      group[i] = (int)(Math.random()*numIntegers);
    }
    mergesort(group);

    for (int i = 0; i < 10; i++) {
      System.out.println(group[i]);
    }
  }

  static void mergesort (int[] A) {
    int[] copy = Arrays.copyOf(A, A.length);
    mergesort (copy, A, 0, A.length);
  }

  static void mergesort(int[] A, int[] result, int start, int end) {
    if (end - start < 2) {
      return;
    }

    if (end - start == 2) {
      if (result[end-2] > result[end-1]) {
        int tmp = result[end-2];
        result[end-2] = result[end-1];
        result[end-1] = tmp;
      }
      return;
    }

    int mid = (end + start)/2;
    mergesort(result, A, start, mid);
    mergesort(result, A, mid, end);

    for (int i = start, j = mid, idx=start; idx < end; idx++) {
      if (j >= end || (i < mid && A[i] < A[j])) {
        result[idx] = A[i++];
      } else {
        result[idx] = A[j++];
      }
    }
  }
}
```

Run this code; you see the first ten randomly generated integers in sorted order. Let's review this code more closely:

OBSERVE: MergeSort invocation

```
  static void mergesort (int[] A) {
    int[] copy = Arrays.copyOf(A, A.length);
    mergesort (copy, A, 0, A.length);
  }
```

To sort the array, we make a full copy and then internally invoke **mergesort** to sort the copy with **A** as the ultimate destination. Note that the arguments to pass in are **0** and **A.length**, which reflect the index values into A, namely

inclusive on the left side with **0** and exclusive on the right side with **A.length**.

All logic once again resides in the recursive method. Let's review the base cases:

OBSERVE: Recursive base case of MergeSort

```
static void mergesort(int[] A, int[] result, int start, int end) {
  if (end - start < 2) {
    return;
  }

  if (end - start == 2) {
    if (result[end-2] > result[end-1]) {
      int tmp = result[end-2];
      result[end-2] = result[end-1];
      result[end-1] = tmp;
    }
    return;
  }
```

If end - start is less than 2, there is either no element or a single element to be sorted, which means nothing needs to be done. When **end-start equals 2**, there are two elements to be sorted. This code executes only as a base case in the recursion, which means that it's the first time the method is inspecting the array subrange of **[start,end)**. Because the result must be stored in the **result** array, this code reorders the values it finds there.

The final elements in **mergesort** show how to merge the sorted left and right sub-arrays:

OBSERVE: Merging in O(n) time

```
  int mid = (end + start)/2;
  mergesort(result, A, start, mid);
  mergesort(result, A, mid, end);

  for (int i = start, j = mid, idx=start; idx < end; idx++) {
    if (j >= end || (i < mid && A[i] < A[j])) {
      result[idx] = A[i++];
    } else {
      result[idx] = A[j++];
    }
  }
```

This code first **recursively sorts the left half and right half of the range [start, end)**, placing the properly ordered elements in the array referenced as **A**. Then it uses two indices, **i** and **j**, to iterate over each of these sub-ranges, always copying the smaller of A[i] and A[j] into the properly located **result[idx]**. There are three cases to consider:

1. The right side is exhausted (**j >= end**), in which case you can grab the remaining elements from **A[i]**.
2. The left side is exhausted (**i >= mid**), in which case you can grab the remaining elements from **A[j]**.
3. The left and right side have elements; if **A[i] < A[j]**, insert **A[i]**, otherwise insert **A[j]**.

Once the for loop completes, **result** has the merged (and sorted) elements from the subarray **[start, end)** of the original array A.

Working with Large Datasets

You are going to use *MergeSort* to sort large collections of values. You're going to need additional storage for that to work. You don't actually need to store the entire collection in main memory to sort its contents. Let's start by defining the problem instance. The input of *n* integers will be stored in a binary file containing *4*n* bytes. Practice using this structure by writing this sample program:

In the **sort** package of the **/src** source folder, create a class **BinaryIntegerFile** as shown:

CODE TO TYPE: BinaryIntegerFile

```
package sort;

import java.io.*;

public class BinaryIntegerFile {
  public static void main(String[] args) throws IOException {
    int numIntegers = 4096;
    File f = new File ("IntegerFile.bin");
    DataOutputStream dos = new DataOutputStream(new FileOutputStream(f));
    for (int i = 0; i < numIntegers; i++) {
       dos.writeInt((int)(Math.random()*numIntegers));
    }
    dos.close();

    DataInputStream dis = new DataInputStream(new FileInputStream(f));

    System.out.println("First five sorted numbers");
    for (int i = 0; i < 5; i++) {
      System.out.println(dis.readInt());
    }
    dis.skipBytes(4*(numIntegers-10));
    System.out.println("Last five sorted numbers");
    for (int i = 0; i < 5; i++) {
      System.out.println(dis.readInt());
    }
    dis.close();
  }
}
```

Run this program; you'll see something like this (your numbers will be different because they are randomly generated):

INTERACTIVE SESSION: Output from BinaryIntegerFile. Note that sort function is not yet implemented.

```
First five sorted numbers
2935
3918
245
2885
2496
Last five sorted numbers
2748
3716
1972
2086
1350
```

Let's take a closer look at this code. The **java.io** package contains a number of classes to read and write information to the file system. The fundamental abstraction is a *stream*, which represents a sequence of data. An **InputStream** reads data from a source and an **OutputStream** writes data to a source. In the code, a **DataInputStream is used to read primitive Java data types from the input stream (such as int and float values)** while a **DataOutputStream writes primitive Java data types to an output stream**.

OBSERVE: Creating a random binary file of integers

```
  int numIntegers = 4096;
  File f = new File ("IntegerFile.bin");
  DataOutputStream dos = new DataOutputStream(new FileOutputStream(f));
  for (int i = 0; i < numIntegers; i++) {
    dos.writeInt((int)(Math.random()*numIntegers));
  }
  dos.close();
```

Using a **DataOutputStream** object, the above code writes 4,096 integers in binary format to the file and closes it. Once created, this file will contain 16,384 bytes because the integers are written in binary format where each integer value requires four bytes. Don't bother trying to open this file in Eclipse because the data is stored in binary format so Eclipse will just present you with the raw data. You can retrieve the integer values that were stored using **DataInputStream**, which properly decodes the binary formatted encoding of the integer values in the file. The second part of the code reads in this file and prints the first five integers and the last five integers in the file.

OBSERVE: Read integers from file

```
    DataInputStream dis = new DataInputStream(new FileInputStream(f));

    System.out.println("First five sorted numbers");
    for (int i = 0; i < 5; i++) {
      System.out.println(dis.readInt());
    }
    dis.skipBytes(4*(numIntegers-10));
    System.out.println("Last five sorted numbers");
    for (int i = 0; i < 5; i++) {
      System.out.println(dis.readInt());
    }
    dis.close();
```

This code uses a **DataInputStream** to retrieve the values from the file. Note that it reads the first five integers and then **skips the requisite number of bytes** (there are four bytes for each integer) so it can then read the last five numbers from the file. Clearly this file isn't sorted; you'll solve this by implementing a *MergeSort* that operates over a **File** containing integer values, rather than an in-memory array of integer values.

To make this work, you have to access a file in the same way that you would otherwise access an array. You know the structure of *MergeSort* from the implementation you completed earlier, all you need to do now is map those concepts to a file. Consider using **RandomAccessFile**, provided by the **java.io** package, which allows you to access any byte within a file randomly. Knowing that the file contains a collection of integers in 4-byte format, you can determine that to read the *nth* int value from the file, you need to start reading 4 bytes from position *n*4*. Similar logic is used to write an integer to replace the *nth* int value in the file. All of these operations will succeed with index values of type **long**, which means you can process extremely large files if you want.

 In the **sort** package of the **/src** source folder, create a **MergeSortFile** class as shown:

CODE TO TYPE: MergeSortFile class

```
package sort;

import java.io.*;

public class MergeSortFile {

  static void mergesort (File A) throws IOException {
    File copy = new File (A.getPath() + ".tmp");
    copyFile(A, copy);

    // TBA: invoke MergeSort

  }

  static void copyFile(File src, File dest) throws IOException {
    FileInputStream fis = new FileInputStream(src);
    FileOutputStream fos = new FileOutputStream (dest);
    byte[] bytes = new byte[4*1048576];
    int numRead;
    while ((numRead = fis.read(bytes)) > 0) {
      fos.write(bytes, 0, numRead);
    }
    fis.close();
    fos.close();
  }
}
```

The **mergesort** method prepares for the algorithm by making a full copy of the source file, **A**. For demonstration purposes, the file copy is created in your workspace, but normally you would use the static method

File.createTempFile instead to create a temporary file in the default temporary directory. The **copyFile** method copies bytes in chunks of four megabytes to replicate the file. To test out the above code, modify **BinaryIntegerFile** to use the **mergesort** method in **MergeSortFile**.

CODE TO TYPE: Modified BinaryIntegerFile

```
package sort;

import java.io.*;

public class BinaryIntegerFile {
  public static void main(String[] args) throws IOException {
    int numIntegers = 4096;
    File f = new File ("IntegerFile.bin");
    DataOutputStream dos = new DataOutputStream(new FileOutputStream(f));
    for (int i = 0; i < numIntegers; i++) {
       dos.writeInt((int)(Math.random()*numIntegers));
    }
    dos.close();

    long now = System.currentTimeMillis();
    MergeSortFile.mergesort(f);
    System.out.println((System.currentTimeMillis() - now) + " ms.");

    DataInputStream dis = new DataInputStream(new FileInputStream(f));

    System.out.println("First five sorted numbers");
    for (int i = 0; i < 5; i++) {
      System.out.println(dis.readInt());
    }
    dis.skipBytes(4*(numIntegers-10));
    System.out.println("Last five sorted numbers");
    for (int i = 0; i < 5; i++) {
      System.out.println(dis.readInt());
    }
    dis.close();
  }
}
```

Now execute **BinaryIntegerFile** and refresh your workspace. You will see two top-level files: **IntegerFile.bin** and **IntegerFile.bin.tmp**. Select both of these files in the Java package browser (holding down the **Shift** key and click each icon), and right-click on either file to select **Compare With | Each Other**. The files are identical, because you haven't yet written any code to sort the data.

You are now ready to complete the *MergeSort* implementation. Modify **MergeSortFile** as follows:

Modified MergeSortFile

```
package sort;

import java.io.*;

public class MergeSortFile {

  static void mergesort (File A) throws IOException {
    File copy = new File (A.getPath() + ".tmp");
    copyFile(A, copy);

    // TBA: invoke MergeSort
    RandomAccessFile src = new RandomAccessFile(A, "rw");
    RandomAccessFile dest = new RandomAccessFile(copy, "rw");

    mergesort (dest, src, 0,  A.length());
    src.close();
    dest.close();
    copy.delete();
  }

  static void copyFile(File src, File dest) throws IOException {
    FileInputStream fis = new FileInputStream(src);
    FileOutputStream fos = new FileOutputStream (dest);
    byte[] bytes = new byte[4*1048576];
    int numRead;
    while ((numRead = fis.read(bytes)) > 0) {
      fos.write(bytes, 0, numRead);
    }
    fis.close();
    fos.close();
  }

  static void mergesort(RandomAccessFile A, RandomAccessFile result,
                        long start, long end) throws IOException {

    if (end - start < 8) {
      return;
    }

    if (end - start == 8) {
      result.seek(end-8);
      int left = result.readInt();
      int right = result.readInt();
      if (left > right) {
        result.seek(end-8);
        result.writeInt(right);
        result.writeInt(left);
      }
      return;
    }

    long mid = (end + start)/8*4;
    mergesort(result, A, start, mid);
    mergesort(result, A, mid, end);

    result.seek(start);
    for (long i = start, j = mid, idx=start; idx < end; idx += 4) {
      A.seek(i);
      int Ai = A.readInt();
      int Aj = 0;
      if (j < end) { A.seek(j); Aj = A.readInt(); }
      if (j >= end || (i < mid && Ai < Aj)) {
        result.writeInt(Ai);
        i += 4;
      } else {
        result.writeInt(Aj);
        j += 4;
```

```
      }
    }
  }
}
```

The modified **mergesort** method now opens two **RandomAccessFile** objects on the two files and they are both opened in *read/write* mode because they will both be updated during the *MergeSort* algorithm. The **mergesort** method is invoked by requesting to sort the contents of the copied file into the original file. Once done, the copied file can be deleted.

The mergesort(RandomAccessFile A, RandomAccessFile result, long start, long end) method performs the recursive *MergeSort* of the given range **[start, end)** of the underlying files. These parameters are both of type **long** to enable this method to sort files that can be several gigabytes in size. For this lesson, the files will only be several megabytes in size; feel free to generate files of this size on your home computer!

The structure of this method follows the earlier examples.

Go back and execute **BinaryIntegerFile** and the output will appear properly (though your random numbers will be different):

Output from BinaryIntegerFile

```
First five sorted numbers
1
3
4
6
6
Last five sorted numbers
4091
4091
4093
4093
4095
```

Refresh the files in your workspace; the temporary file used during the sort has been deleted. Let's take a closer look at this code. First, let's inspect the base cases of the recursion:

OBSERVE: mergesort base cases for recursion

```
  static void mergesort(RandomAccessFile A, RandomAccessFile result,
                        long start, long end) throws IOException {

    if (end - start < 8) {
      return;
    }

    if (end - start == 8) {
      result.seek(end-8);
      int left = result.readInt();
      int right = result.readInt();
      if (left > right) {
        result.seek(end-8);
        result.writeInt(right);
        result.writeInt(left);
      }
      return;
    }
```

Recall that integers are stored using 4 bytes. The offsets **start** and **end** are index locations within the **RandomAccessFile**; in addition, **start** and **end** must be evenly divisible by 4. The condition **end - start < 8** determines when the subrange **[start, end)** contains zero or one element; when this occurs, no sorting needs to take place and the method can simply return.

The second base case needs to swap the two neighboring integers in **result** if they are out of order. When **end - start == 8** you know that **[start, end)** contains two elements exactly. The above code uses the **seek** method to find that location in the file for the first of these two integers. It then reads in two integers sequentially from the 8 bytes

stored at that position in that file. If these two numbers are out of order, it goes back to the beginning of that range in the file and writes the two integers in the proper order.

The final case demonstrates how to merge the two sub-ranges together. It starts with a little mathematical optimization. Mergesort must divide the range into two parts, but each sub-range must contain a number of bytes that is divisible by four. For example, if the range contained 7 integers for a total of 28 bytes, it might be represented as **[0,28)**. Simply dividing **(0+28)/2** would give 14, which is not divisible by 4. Instead, divide **(0+28)/8** (to get 3 using integer division) and then multiply by 4 to get 12, which is roughly half of the range.

OBSERVE: Merging case in MergeSort

```
long mid = (end + start)/8*4;
mergesort(result, A, start, mid);
mergesort(result, A, mid, end);

result.seek(start);
for (long i = start, j = mid, idx=start; idx < end; idx += 4) {
  A.seek(i);
  int Ai = A.readInt();
  int Aj = 0;
  if (j < end) { A.seek(j); Aj = A.readInt(); }
  if (j >= end || (i < mid && Ai < Aj)) {
    result.writeInt(Ai);
    i += 4;
  } else {
    result.writeInt(Aj);
    j += 4;
  }
}
```

The above code recursively **invokes mergesort on the left and right sub-ranges**, after which the file on disk referenced by **A** will contain the two sorted sub-ranges waiting to be merged. The code takes advantage of a nice optimization in that the integers written to **result** will be written sequentially, so it only needs to **seek to the starting location** of that output sequence in **result** before starting the **for** loop. When the code determines the **Ai** and **Aj** values, it must seek the proper file position within **A** and then read the integer encoded there.

Note that, in this **for** loop, the index values *i*, *j*, and *idx* are all incremented by 4 because they reference positions inside the file that contains the 4-byte integer encodings.

The condition **(j >= end || (i < mid && Ai < Aj))** takes advantage of "short-circuit" logical evaluation. That is, if **j >= end**, the second part of the condition (after the "||") is not executed. However, if **j < end**, you can retrieve **Aj** from the file. For this reason, the code first loads up **Aj** if it exists to prepare for the short-circuit conditional.

Never Be Satisfied

Despite the success of the code, it still feels like it takes too long to complete. The problem is likely that as the files get larger, the number of disk accesses begins to dominate the performance of the algorithm. Fortunately there is a "drop-in replacement" for file access based on an operating-system capability known as "Memory Mapped Files." The implementation is found in the **java.nio** package, known as the "new input/output" Java package that contains many high-performance classes.

 In the **sort** package of the **/src** source folder, create a class named **MergeSortFileMapped**. Much of this code will be familiar to you because it follows the exact implementation style of the earlier MergeSort.

CODE TO TYPE: MergeSortFileMapped class

```
package sort;

import java.io.*;
import java.nio.*;
import java.nio.channels.*;

public class MergeSortFileMapped {

  static void mergesort (File A) throws IOException {
    File copy = File.createTempFile("Mergesort", ".bin");
    MergeSortFile.copyFile(A, copy);

    RandomAccessFile src = new RandomAccessFile(A, "rw");
    RandomAccessFile dest = new RandomAccessFile(copy, "rw");
    FileChannel srcC = src.getChannel();
    FileChannel destC = dest.getChannel();
    MappedByteBuffer srcMap = srcC.map(FileChannel.MapMode.READ_WRITE, 0, src.length());
    MappedByteBuffer destMap = destC.map(FileChannel.MapMode.READ_WRITE, 0, dest.length()

    mergesort (destMap, srcMap, 0, (int) A.length());
    src.close();
    dest.close();
  }

  static void mergesort(MappedByteBuffer A, MappedByteBuffer result,
                        int start, int end) throws IOException {

    if (end - start < 8) {
      return;
    }

    if (end - start == 8) {
      result.position(start);
      int left = result.getInt();
      int right = result.getInt();
      if (left > right) {
        result.position(start);
        result.putInt(right);
        result.putInt(left);
      }
      return;
    }

    int mid = (end + start)/8*4;
    mergesort(result, A, start, mid);
    mergesort(result, A, mid, end);

    result.position(start);
    for (int i = start, j = mid, idx=start; idx < end; idx += 4) {
      int Ai = A.getInt(i);
      int Aj = 0;
      if (j < end) { Aj = A.getInt(j); }
      if (j >= end || (i < mid && Ai < Aj)) {
        result.putInt(Ai);
        i += 4;
      } else {
        result.putInt(Aj);
        j += 4;
      }
    }
  }
}
```

To execute this code instead of the earlier version, modify **BinaryIntegerFile** as shown:

CODE TO TYPE: Modifications to BinaryIntegerFile

```
package sort;

import java.io.*;

public class BinaryIntegerFile {
  public static void main(String[] args) throws IOException {
    int numIntegers = 655364096;
    File f = new File ("IntegerFile.bin");
    DataOutputStream dos = new DataOutputStream(new FileOutputStream(f));
    for (int i = 0; i < numIntegers; i++) {
      dos.writeInt((int)(Math.random()*numIntegers));
    }
    dos.close();

    long now = System.currentTimeMillis();
    MergeSortFile.mergesort(f);
    MergeSortFileMapped.mergesort(f);
    System.out.println((System.currentTimeMillis() - now) + " ms.");

    DataInputStream dis = new DataInputStream(new FileInputStream(f));
    System.out.println("First five sorted numbers");
    for (int i = 0; i < 5; i++) {
      System.out.println(dis.readInt());
    }
    dis.skipBytes(4*(numIntegers-10));
    System.out.println("Last five sorted numbers");
    for (int i = 0; i < 5; i++) {
      System.out.println(dis.readInt());
    }
    dis.close();
  }
}
```

Run it to sort just over 65,000 integer values. The code spends most of its time writing the random numbers to the disk file to prepare for the algorithm. The execution now takes far less time (32 milliseconds instead of 10276 milliseconds). This code executes so much faster because when you're working with data on disk, you need to limit the frequency of disk access to maximize the efficiency of your code. Inside the Java Virtual Machine, the **java.nio** package is integrated with the virtual memory manager of the operating system. Memory-mapped files are loaded into memory one entire page at a time, and each operating system is fine-tuned so these operations execute as efficiently as possible. When you modify information in a memory-mapped file, it will be written out to the file one page at a time; the operating system is responsible for carrying this operation out efficiently as well. Now your program is no longer in charge of reading and writing bytes from a file directly; it updates memory directly, as managed by the **MappedByteBuffer** class. Ultimately this class determines when the updated memory is written to the file.

Let's review this code:

OBSERVE: Using MappedByteBuffers to access file data

```
  static void mergesort (File A) throws IOException {
    File copy = File.createTempFile("Mergesort", ".bin");
    MergeSortFile.copyFile(A, copy);

    RandomAccessFile src = new RandomAccessFile(A, "rw");
    RandomAccessFile dest = new RandomAccessFile(copy, "rw");
    FileChannel srcC = src.getChannel();
    FileChannel destC = dest.getChannel();
    MappedByteBuffer srcMap = srcC.map(FileChannel.MapMode.READ_WRITE, 0, src.length());
    MappedByteBuffer destMap = destC.map(FileChannel.MapMode.READ_WRITE, 0, dest.length()

    mergesort (destMap, srcMap, 0,  (int) A.length());
    src.close();
    dest.close();
  }
```

One unfortunate drawback with using MappedByteBuffer is that on many operating systems (and on Windows in particular) once a file has been mapped, it cannot be deleted from within the Java program. The above code, therefore, **creates a temporary file in the designated default temporary directory** which will eventually be cleaned up by the user. From a **RandomAccessFile**, it is **possible to retrieve its FileChannel descriptor**, which is used to **construct the respective MappedByteBuffer objects**.

Changes to the **mergesort** method are more subtle:

OBSERVE: mergesort revised to use MappedByteBuffer

```
static void mergesort(MappedByteBuffer A, MappedByteBuffer result,
                      int start, int end) throws IOException {

  if (end - start < 8) {
    return;
  }

  if (end - start == 8) {
    result.position(start);
    int left = result.getInt();
    int right = result.getInt();
    if (left > right) {
      result.position(start);
      result.putInt(right);
      result.putInt(left);
    }
    return;
  }

  int mid = (end + start)/8*4;
  mergesort(result, A, start, mid);
  mergesort(result, A, mid, end);

  result.position(start);
  for (int i = start, j = mid, idx=start; idx < end; idx += 4) {
    int Ai = A.getInt(i);
    int Aj = 0;
    if (j < end) { Aj = A.getInt(j); }
    if (j >= end || (i < mid && Ai < Aj)) {
      result.putInt(Ai);
      i += 4;
    } else {
      result.putInt(Aj);
      j += 4;
    }
  }
}
```

start and **end** are int values again. The MappedByteBuffer class only supports integer indexing, which means the files to be sorted cannot be greater than 2^{32} bytes in size (roughly 4 gigabytes).

Let's review the base cases of the recursion:

OBSERVE: MergeSortFileMapped base recursive cases

```
if (end - start < 8) {
  return;
}

if (end - start == 8) {
  result.position(start);
  int left = result.getInt();
  int right = result.getInt();
  if (left > right) {
    result.position(start);
    result.putInt(right);
    result.putInt(left);
  }
  return;
}
```

When asked to sort two elements, the code uses the **getInt** method of the MappedByteBuffer class to retrieve the integer stored at the proper offset of **start**. If the MappedByteBuffer does not have this information in main memory, it will read the information into memory one page at a time. If this memory is updated (using the **putInt** methods) it won't be written to disk until the **MappedByteBuffer** determines that it can be written eficiently. As a programmer, you no longer know whether a **getInt** or **end** method accesses the file system; you can simply program it correctly while leaving MappedByteBuffer responsible for the persistent storage of the information.

OBSERVE: Completing the mergesort

```
int mid = (end + start)/8*4;
mergesort(result, A, start, mid);
mergesort(result, A, mid, end);

result.position(start);
for (int i = start, j = mid, idx=start; idx < end; idx += 4) {
  int Ai = A.getInt(i);
  int Aj = 0;
  if (j < end) { Aj = A.getInt(j); }
  if (j >= end || (i < mid && Ai < Aj)) {
    result.putInt(Ai);
    i += 4;
  } else {
    result.putInt(Aj);
    j += 4;
  }
}
```

Despite the large number of read and write statements that access the **result** and **A** files, the MappedByteBuffer class ensures that these operations act on information stored in main memory. You can't predict when (Java) will write the info to a file, so the MappedByteBuffer class ensures that data is read from memory (instead of the file). It'll be more efficient because it's faster to read from memory than from a file (in this case, almost 300 times faster).

Lessons Learned

Often you can improve the efficiency of an algorithm by storing additional state information. You see this on a small scale in **java.lang.String**, which caches its computed hash value to improve the performance of **hashCode**. Often you can achieve efficient *O(n log n)* performance by storing additional *O(n)* storage information.

To determine the appropriate algorithm to use, be sure to characterize the storage requirements in addition to the run-time performance. In most cases, the additional storage will be *O(n)*, which typically is an acceptable trade-off to make.

Accessing information on disk is typically *thousands of times slower than accessing information in main memory*. When designing algorithms that access data on disk, you must find ways to reduce the number of individual reads and writes, choosing instead to let the operating system optimize input/output access.

Practice some of the things you learned in this lesson in the project. See you in the next lesson!