

COMP 410 - Spring 2014

Programming Assignment 3



PROJECT DESCRIPTION

All the sorting algorithms we have discussed in class require that the input to be sorted fits entirely into main memory prior to being sorted. There are, however, applications where the input is much too large to fit into memory. Such inputs are sorted using what are called **external sorting algorithms**. In this assignment, we will simulate a simple external sorting algorithm that assumes that its input is provided on a *magnetic tape*, which can only be accessed sequentially.

§1. Read Sections 7.12.1 – 7.12.3 of your text (reproduced here in the appendix) and make sure that you understand the simple sorting algorithm using 4 tapes that is described there.

§2. The following class simulates a tape drive:

```
/**
 * This class simulates a tape drive.
 */
public class TapeDrive<T> {

    private T[] tape;
    private int currentPos = 0;

    @SuppressWarnings("unchecked")
    public TapeDrive(int capacity) {
        tape = (T[]) new Object[capacity];
    }

    // Write in a wrap-around manner
    public void write(T t) {
        tape[currentPos] = t;
        currentPos = (currentPos + 1) % tape.length;
    }

    // Read in a wrap-around manner
    public T read() {
        T t = tape[currentPos];
        currentPos = (currentPos + 1) % tape.length;
        return t;
    }

    public void reset() {
        currentPos = 0;
    }
}
```

§3. You are to complete the methods `generateRandomTape` and `sort` within the class `SortOnTapes` outlined below.

```
/**
 * This class represents a machine with limited memory that can sort tape drives.
 */
public class SortOnTapes {

    // Constants. These will be changed when testing your program.
    public static int TapeSize = 100;
    public static int MainMemorySize = 10;

    // Memory to be used when sorting
    private static Integer[] memory = new Integer[MainMemorySize];

    /**
     * This creates a new tape drive of size n and fills it with random integers.
     * @param n The size or capacity of the new drive
     * @return The new TapeDrive containing n random integers
     */
    public static TapeDrive<Integer> generateRandomTape(int n) {
        // TODO: Write this
    }

    /**
     * This sorts a tape drive using an external sorting algorithm. The sorted data
     * should be placed on t1.
     * @param t1 The drive to sort
     */
    public static void sort(TapeDrive<Integer> t1) {
        // Additional tapes to use when sorting.
        TapeDrive<Integer> t2 = new TapeDrive<Integer>(TapeSize);
        TapeDrive<Integer> t3 = new TapeDrive<Integer>(TapeSize);
        TapeDrive<Integer> t4 = new TapeDrive<Integer>(TapeSize);

        // TODO: Write this
    }

    public static void main(String[] args) {
        TapeDrive<Integer> drive = generateRandomTape(TapeSize);
        sort(drive);

        // This tests to see if the drive is, indeed, sorted.
        boolean sorted = true;
        int prev = Integer.MIN_VALUE;
        for (int i = 0; i < TapeSize; i++) {
            int p = prev;
            sorted &= p > (prev = drive.read());
        }
        if (sorted)
            System.out.println("Success! The tape drive is sorted.");
        else
            System.out.println("Failure! The tape drive is not sorted.");
    }
}
```

For this assignment you are to complete the implementation of the `SortOnTapes` class. Specifically,

- 1) Write the method `generateRandomTape()` that creates a new `TapeDrive` and populates it with n random integers.
- 2) Complete the `sort` method, which sorts the provided `TapeDrive` by using three additional drives and the system memory. See the book excerpts for more information on how to go about this. You may declare an additional *constant*¹ number of variables, but no more arrays, linked lists, etc. You may write additional helper methods, define additional classes, etc. But you may NOT modify the `TapeDrive` class in any manner.

NOTE

Do not use any in-built Java implementation (e.g., `Stack`, `Queue`, `Lists`, `ArrayList`, etc) from the `java.util.*` API. You *must* implement the above from scratch. The only exception is `java.util.Random` which you will use when creating a `TapeDrive` filled with random integers.

Credit will be awarded for efficiency in implementation – make your implementation efficient in terms of both run-time and memory usage.

The source code of the above classes will be provided for your convenience.

GRADING RUBRIC

- (15 points) Correct implementation of `generateRandomTape`
- (40 points) Correct implementation of `sort`
- (25 points) Not creating any additional arrays or collections
- (10 points) Efficiency of `sort`
- (10 points) Good, readable code

SUBMISSION INSTRUCTIONS

Upload all your source code in a `.zip` file to Sakai. You are responsible for ensuring that your program compiles and functions properly. Any non-functioning program will receive a zero.

HONOR CODE

Please review the honor code description from the course syllabus. No collaboration (with anyone) is permitted in assignments. Collaboration in assignments, or the use of code not the students own, constitutes an honor code violation. Any violation will be reported to the Student Attorney General.

1. I.e., not depending on the input size.

7.12.1 Why We Need New Algorithms

Most of the internal sorting algorithms take advantage of the fact that memory is directly addressable. Shellsort compares elements $a[i]$ and $a[i-h_k]$ in one time unit. Heapsort compares elements $a[i]$ and $a[i*2+1]$ in one time unit. Quicksort, with median-of-three partitioning, requires comparing $a[\text{left}]$, $a[\text{center}]$, and $a[\text{right}]$ in a constant number of time units. If the input is on a tape, then all these operations lose their efficiency, since elements on a tape can only be accessed sequentially. Even if the data is on a disk, there is still a practical loss of efficiency because of the delay required to spin the disk and move the disk head.

To see how slow external accesses really are, create a random file that is large, but not too big to fit in main memory. Read the file in and sort it using an efficient algorithm. The time it takes to read the input is certain to be significant compared to the time to sort the input, even though sorting is an $O(N \log N)$ operation and reading the input is only $O(N)$.

7.12.2 Model for External Sorting

The wide variety of mass storage devices makes external sorting much more device dependent than internal sorting. The algorithms that we will consider work on tapes, which are probably the most restrictive storage medium. Since access to an element on tape is done by winding the tape to the correct location, tapes can be efficiently accessed only in sequential order (in either direction).

We will assume that we have at least three tape drives to perform the sorting. We need two drives to do an efficient sort; the third drive simplifies matters. If only one tape drive is present, then we are in trouble: Any algorithm will require $\Omega(N^2)$ tape accesses.

7.12.3 The Simple Algorithm

The basic external sorting algorithm uses the merging algorithm from mergesort. Suppose we have four tapes, T_{a1} , T_{a2} , T_{b1} , T_{b2} , which are two input and two output tapes. Depending on the point in the algorithm, the a and b tapes are either input tapes or output tapes. Suppose the data are initially on T_{a1} . Suppose further that the internal memory can hold (and sort) M records at a time. A natural first step is to read M records at a time from the input tape, sort the records internally, and then write the sorted records alternately to T_{b1} and T_{b2} . We will call each set of sorted records a **run**. When this is done, we rewind all the tapes. Suppose we have the same input as our example for Shellsort.

T_{a1}	81	94	11	96	12	35	17	99	28	58	41	75	15
T_{a2}													
T_{b1}													
T_{b2}													

If $M = 3$, then after the runs are constructed, the tapes will contain the data indicated in the following figure.

T_{a1}							
T_{a2}							
T_{b1}	11	81	94	17	28	99	15
T_{b2}	12	35	96	41	58	75	

Now T_{b1} and T_{b2} contain a group of runs. We take the first run from each tape and merge them, writing the result, which is a run twice as long, onto T_{a1} . Recall that merging two sorted lists is simple; we need almost no memory, since the merge is performed as T_{b1} and T_{b2} advance. Then we take the next run from each tape, merge these, and write the result to T_{a2} . We continue this process, alternating between T_{a1} and T_{a2} , until either T_{b1} or T_{b2} is empty. At this point either both are empty or there is one run left. In the latter case, we copy this run to the appropriate tape. We rewind all four tapes and repeat the same steps, this time using the a tapes as input and the b tapes as output. This will give runs of $4M$. We continue the process until we get one run of length N .

This algorithm will require $\lceil \log(N/M) \rceil$ passes, plus the initial run-constructing pass. For instance, if we have 10 million records of 128 bytes each, and four megabytes of internal memory, then the first pass will create 320 runs. We would then need nine more passes to complete the sort. Our example requires $\lceil \log 13/3 \rceil = 3$ more passes, which are shown in the following figures.

T_{a1}	11	12	35	81	94	96	15
T_{a2}	17	28	41	58	75	99	
T_{b1}							
T_{b2}							

T_{a1}												
T_{a2}												
T_{b1}	11	12	17	28	35	41	58	75	81	94	96	99
T_{b2}	15											

T_{a1}	11	12	15	17	28	35	41	58	75	81	94	96	99
T_{a2}													
T_{b1}													
T_{b2}													

7.12.4 Multiway Merge

If we have extra tapes, then we can expect to reduce the number of passes required to sort our input. We do this by extending the basic (two-way) merge to a k -way merge.

Merging two runs is done by winding each input tape to the beginning of each run. Then the smaller element is found, placed on an output tape, and the appropriate input