Data Structures and Algorithms

BSc. In Computing

Semester 5 Lecture 9

Lecturer: Dr. Simon McLoughlin

- Sorting Algorithms and their efficiency.
  - Bubble Sort
  - Selection Sort
  - Merge Sort
- Things to do…

Sorting Algorithms and their efficiency.

Sorting is a process that organises a collection of data into either ascending or descending order.

Very often we need to sort data before processing in some manner, i.e. searching for customer records.

When the collection of data is large, an efficient method for searching is desireable, i.e, a binary search…. But this requires that the data be sorted.

The data items to be sorted might be integers, characters, strings, or objects.

If each object contains several data fields, we must know which data fields determine the order of the entire object within the collection of data.

The data field is the sort key.

If the objects represent people, for example, you might want to sort on their names, ages, etc.

The sorting algorithms will only sort objects based on their sort key.

We will generally assume that the data items are instances of a class that has implemented the **Comparable** interface.

The Comparable interface method **compareTo** returns either a negative integer, zero, or a positive integer based on whether the sort key is less than, equal to, or greater than the sort key of the specified object.

Bubble Sort

The bubble sort compares adjacent items and exchanges them if they are out of order. This sort usually requires several passes over the data.

- During the first pass, we compare the first two items in the array.
- If they are out of order, we exchange them.

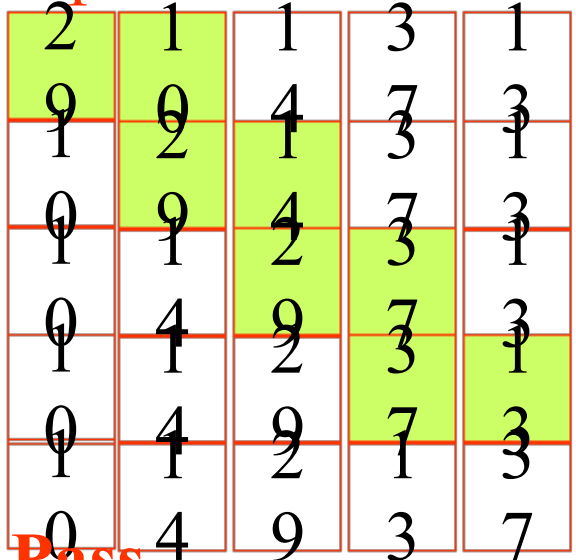
We then compare the items in the next pair, that is, positions 2 and 3 in the array.

- If they are out of order, we exchange them.

We proceed, comparing and exchanging items two at a time until the end of the array is reached.

The first two passes of a bubble sort of an array of five integers

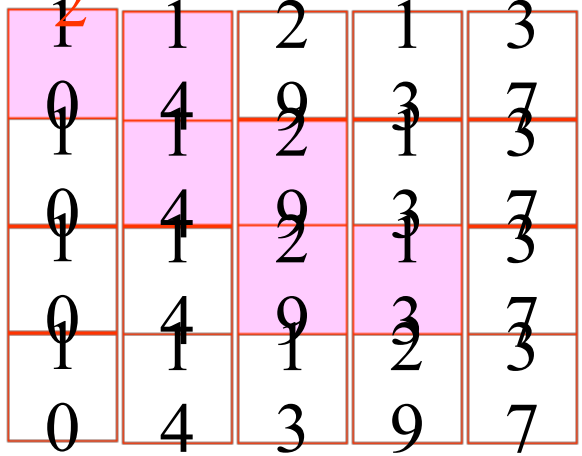
Pass 1



In the diagram we see that we compare the first pair of array cells – 29 and 10 – and exchange them because they are out of order.

Although the array is not sorted after the first pass, the largest item has "bubbled" to its proper position at the end of the array.

Pass 2



During the next pass, we return to the start of the array and consider pairs of items in exactly the same manner.

We do not want to include the last – and largest – item of the array.

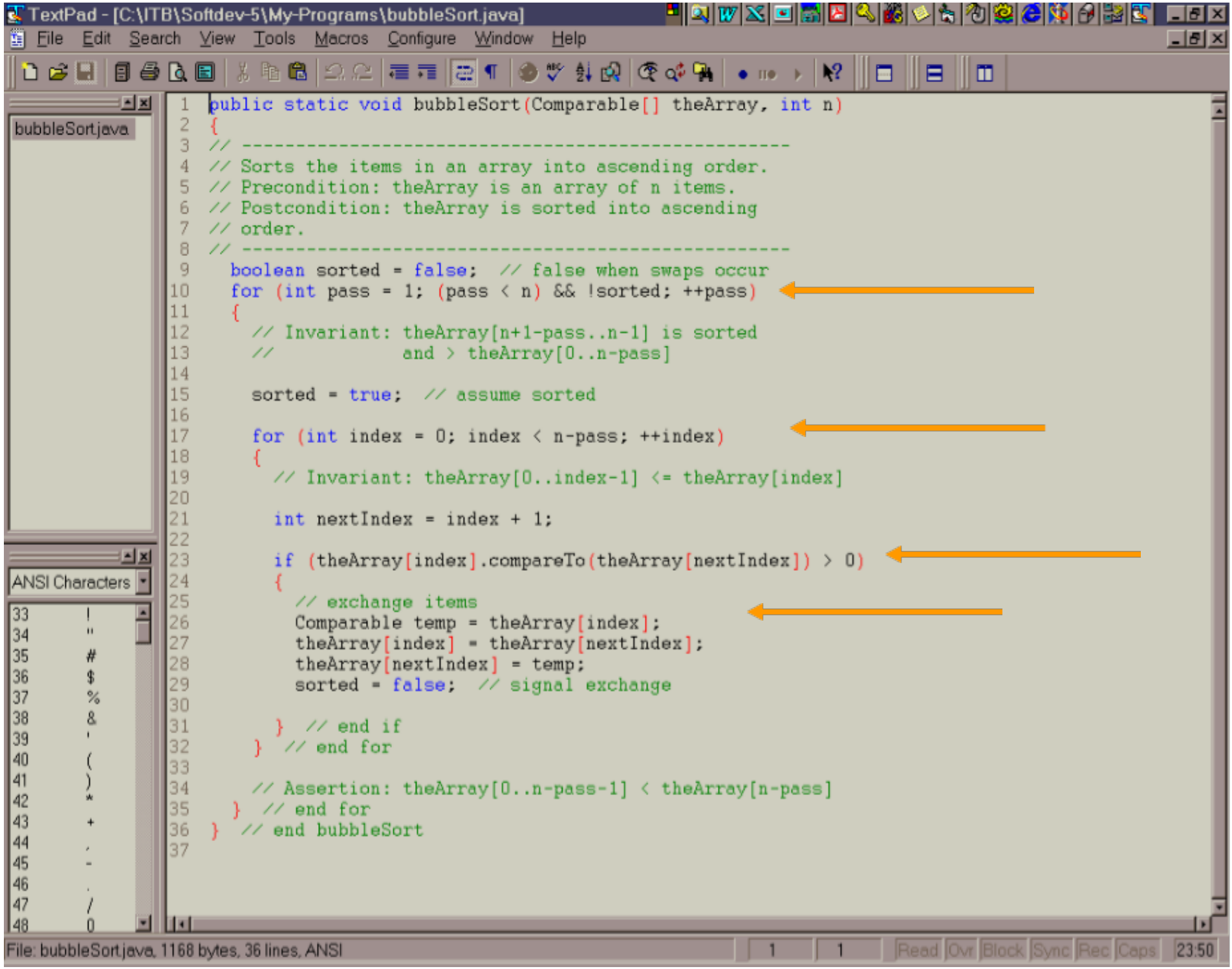
After the second pass the second largest item of the array will be in its proper place in the next to last position of the array.

Ignoring the last two items we proceed with the third pass…. etc., until the array is sorted

- Although a bubble sort requires at most n – 1 passes to sort the array, fewer passes might be possible to sort a particular array.

- We could therefore terminate the process if no exchanges occur during any pass.

The following Java method **bubbleSort** uses a flag to signal when an exchange occurs during a particular pass.


TextPad - [C:\ITB\Softdev-5\My-Programs\bubbleSort.java]
File Edit Search View Tools Macros Configure Window Help
bubbleSort.java
ANSI Characters
33 ! 34 " 35 # 36 $ 37 % 38 & 39 ' 40 (41) 42 * 43 + 44 , 45 - 46 . 47 / 48 0
1 public static void bubbleSort(Comparable[] theArray, int n)
2 {
3 // --
4 // Sorts the items in an array into ascending order.
5 // Precondition: theArray is an array of n items.
6 // Postcondition: theArray is sorted into ascending
7 // order.
8 // --
9 boolean sorted = false; // false when swaps occur
10 for (int pass = 1; (pass < n) && !sorted; ++pass)
11 {
12 // Invariant: theArray[n+1-pass..n-1] is sorted
13 // and > theArray[0..n-pass]
14
15 sorted = true; // assume sorted
16
17 for (int index = 0; index < n-pass; ++index)
18 {
19 // Invariant: theArray[0..index-1] <= theArray[index]
20
21 int nextIndex = index + 1;
22
23 if (theArray[index].compareTo(theArray[nextIndex]) > 0)
24 {
25 // exchange items
26 Comparable temp = theArray[index];
27 theArray[index] = theArray[nextIndex];
28 theArray[nextIndex] = temp;
29 sorted = false; // signal exchange
30
31 } // end if
32 } // end for
33
34 // Assertion: theArray[0..n-pass-1] < theArray[n-pass]
35 } // end for
36 } // end bubbleSort
37
File: bubbleSort.java, 1168 bytes, 36 lines, ANSI
1 1 Read Ovr Block Sync Rec Caps 23:50

Analysis of bubbleSort

The bubble sort requires at most n– 1 passes through the array.

Pass 1 requires n – 1 comparisons and at most n– 1 exchanges.

Pass 2 requires n – 2 comparisons and at most n– 2 exchanges.

In general, …
pass k requires n – k comparisons and at most n– k exchanges.

Therefore, in the worst case, a bubble sort will require a total of:

$(n-1)+(n-2)+\ldots\ldots+1 =$ **$(n^2-n)/2$ comparisons**

and the same number of exchanges. = **$(n^2-n)/2$ exchanges**

The bubble sort is therefore $\mathbf{O(n^2)}$ in the worst case.

The best case occurs when the original data is already sorted – bubble sort uses one pass during which n – 1 comparisons and no exchanges are made.

The bubble sort is $\mathbf{O(n)}$ in the best case.

Selection Sort

To sort using this technique, select a data item and put it in its correct place, select the next item and put it in its place, etc.

This sort is analogous to looking at an entire deck of cards and ordering it by selecting cards one at a time in their proper order.

All the data is essentially visible at the same time!

The selection sort formalises this idea.

To sort an array into ascending order, you first search it for the largest item.

Because we want the last item to be the last in the array, we swap the last array item with the array largest item….even if they are the identical.

We now ignore the last array cell and search the array for the (new) largest data item, and swap this for the second to last array item….and so on.

We continue this selecting and swapping n – 1 times for the n items in the array.

The remaining item that is now in the first position of the array is in its proper place and need not be moved.

A Selection sort of an array of five elements

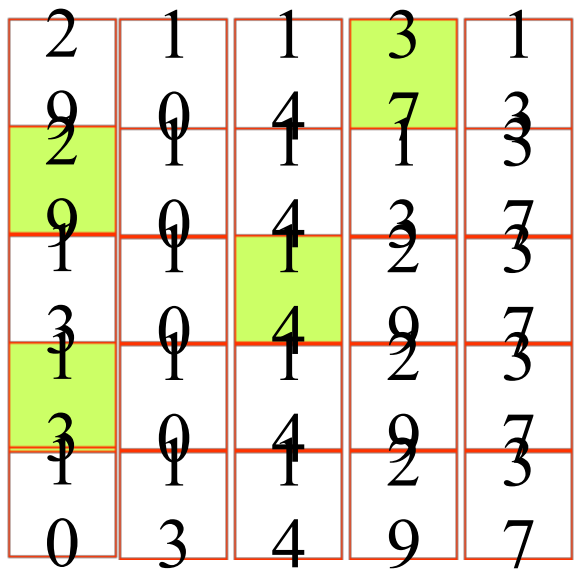
Initial array

After 1st swap

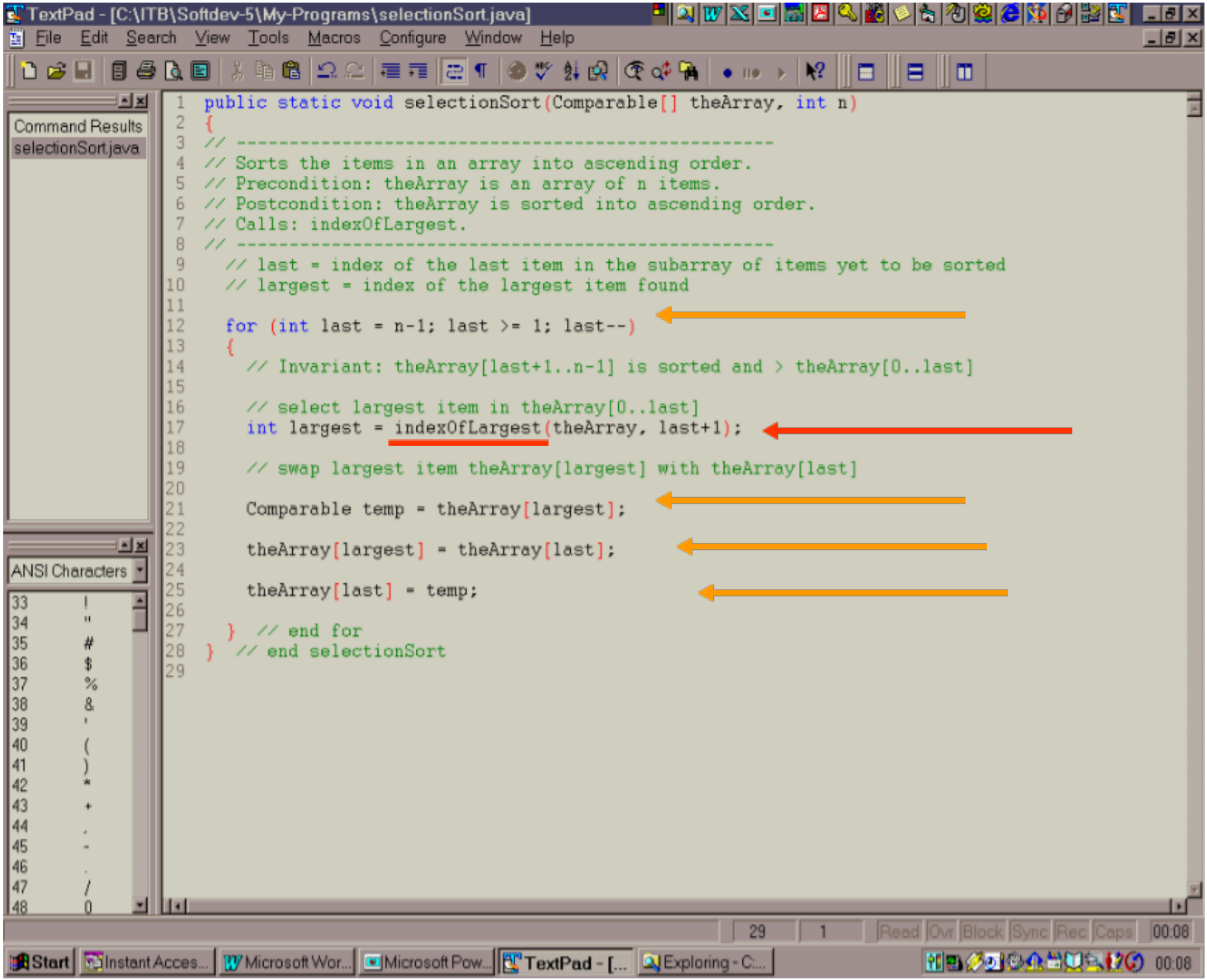
After 2nd swap

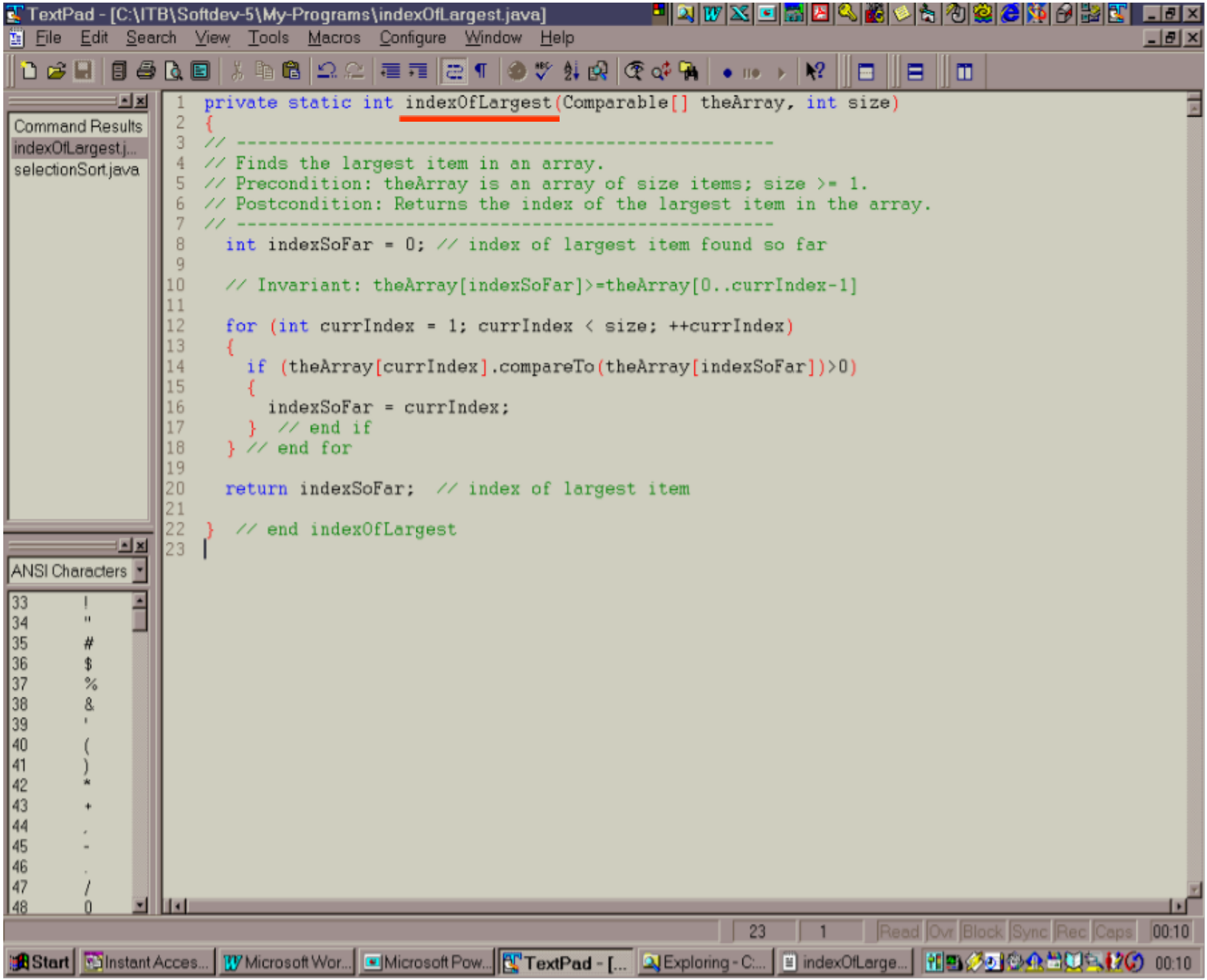
After 3rd swap

After 4th swap



Shaded elements are selected
boldface elements are in order


TextPad - [C:\ITB\Softdev-5\My-Programs\selectionSort.java]
File Edit Search View Tools Macros Configure Window Help
Command Results
selectionSort.java
ANSI Characters
public static void selectionSort(Comparable[] theArray, int n)
{
// --
// Sorts the items in an array into ascending order.
// Precondition: theArray is an array of n items.
// Postcondition: theArray is sorted into ascending order.
// Calls: indexOfLargest.
// --
// last = index of the last item in the subarray of items yet to be sorted
// largest = index of the largest item found
for (int last = n-1; last >= 1; last--)
{
// Invariant: theArray[last+1..n-1] is sorted and > theArray[0..last]
// select largest item in theArray[0..last]
int largest = indexOfLargest(theArray, last+1);
// swap largest item theArray[largest] with theArray[last]
Comparable temp = theArray[largest];
theArray[largest] = theArray[last];
theArray[last] = temp;
} // end for
} // end selectionSort
29 1 Read Ovr Block Sync Rec Caps 00:08
Start Instant Acces... Microsoft Wor... Microsoft Pow... TextPad - [... Exploring - C:... 00:08


TextPad - [C:\ITB\Softdev-5\My-Programs\indexOfLargest.java]
File Edit Search View Tools Macros Configure Window Help
Command Results
indexOfLargest.j...
selectionSort.java
ANSI Characters
33 !
34 "
35 #
36 $
37 %
38 &
39 '
40 (
41)
42 *
43 +
44 ,
45 -
46 .
47 /
48 0
1 private static int indexOfLargest(Comparable[] theArray, int size)
2 {
3 // ---
4 // Finds the largest item in an array.
5 // Precondition: theArray is an array of size items; size >= 1.
6 // Postcondition: Returns the index of the largest item in the array.
7 // ---
8 int indexSoFar = 0; // index of largest item found so far
9
10 // Invariant: theArray[indexSoFar]>=theArray[0..currIndex-1]
11
12 for (int currIndex = 1; currIndex < size; ++currIndex)
13 {
14 if (theArray[currIndex].compareTo(theArray[indexSoFar])>0)
15 {
16 indexSoFar = currIndex;
17 } // end if
18 } // end for
19
20 return indexSoFar; // index of largest item
21
22 } // end indexOfLargest
23
23 1 Read Ovr Block Sync Rec Caps 00:10
Start Instant Acces... Microsoft Wor... Microsoft Pow... TextPad - [... Exploring - C:... indexOfLarge... 00:10

Analysis of Selection Sort

Sorting involves compares, exchanges or moving items.

The comparison operation is typically more involved since actual data values must be compared. In Java, comparison is done by a call to either **equals** or **compareTo**.

Even in cases where the data objects are large and complex, the actual comparison is based on a portion of the object, that is, the sort key.

The **for** loop in the method **selectionSort** executes n – 1 times.

The **selectionSort** calls the method **indexOfLargest** n – 1 times.

Each call to **indexOfLargest** causes its loop to execute **last** times, that is, size – 1.

The n – 1 calls to **indexOfLargest** for values of **last** that range from n – 1 down to 1 mean the number of passes through the loop in **indexOfLargest** is:

$$(n-1)+(n-2)+\ldots+1 = \boxed{(n^2-n)/2 \text{ times.}}$$

Because each execution of **indexOfLargest's** loop performs one comparison, the calls to **indexOfLargest** require $(n^2-n)/2$ comparisons.

At the end of the for loop in **selectionSort**, an exchange is performed between elements theArray[largest] and theArray[last].

Each exchange requires three assignments, or **3 * (n – 1)** data moves.

Together, a selection sort of n items requires:

$(n^2-n)/2 + 3 * (n-1)$ major operations.

By applying the properties of growth rate functions we can ignore the lower order terms to get $O(n^2/2)$ and then ignore the multiplier ½ to get $O(n^2)$.

The selection sort is therefore $O(n^2)$.

- The selection sort is appropriate for only for small n because of the rapid growth of $O(n^2)$.
- While the algorithm requires $O(n^2)$ comparisons, it only requires $O(n)$ data moves.

This algorithm would therefore a reasonable choice in an environment when data moves are very costly, but comparisons are not.

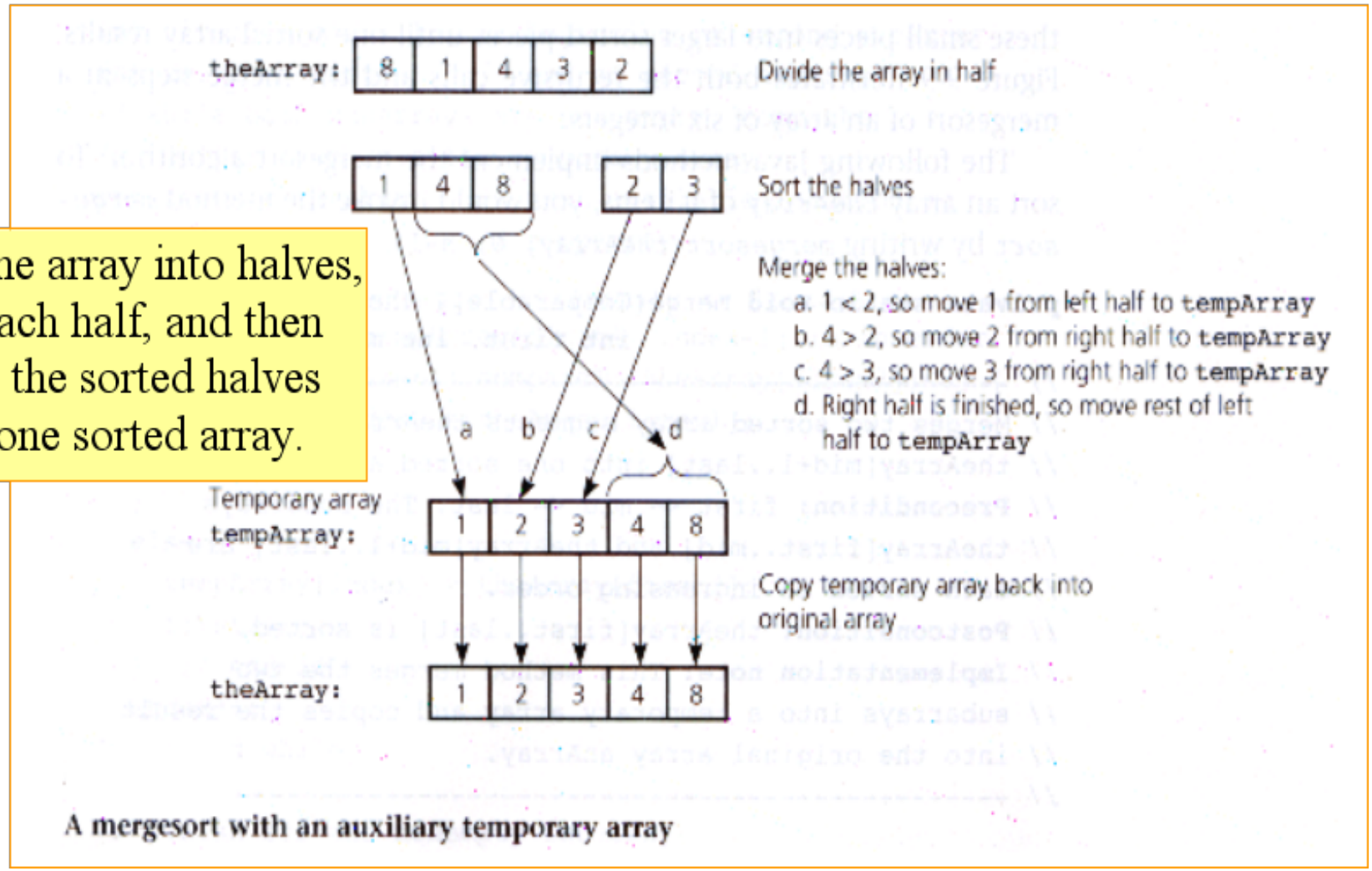
In Java, there is really no such thing as a costly data move in a normal sorting situation because the references and not entire objects are copied in the move operation.

Merge Sort

Mergesort is a divide and conquer sorting algorithms that has a formulation that is highly efficient.

We will find it convenient to express the algorithms in terms of the array theArray[first…last].

Divide the array into halves, sort each half, and then merge the sorted halves into one sorted array.



A mergesort with an auxiliary temporary array

In the diagram the halves <1,4,8> and <2,3> are merged to form the array <1,2,3,4,8>

This merge step compares an item in one half of the array with an item in the other half and moves the smaller item to a temporary array.

This process continues until there are no more items to consider in one half.

- At that time, we simply move the remaining items to the temporary array.
- Finally, we copy the temporary array back into the original array.

How do we sort each half of the array?

Merge sort sorts each half by calling mergesort,

that is, by calling its elf recursively .

The pseudocode for mergeSort is:

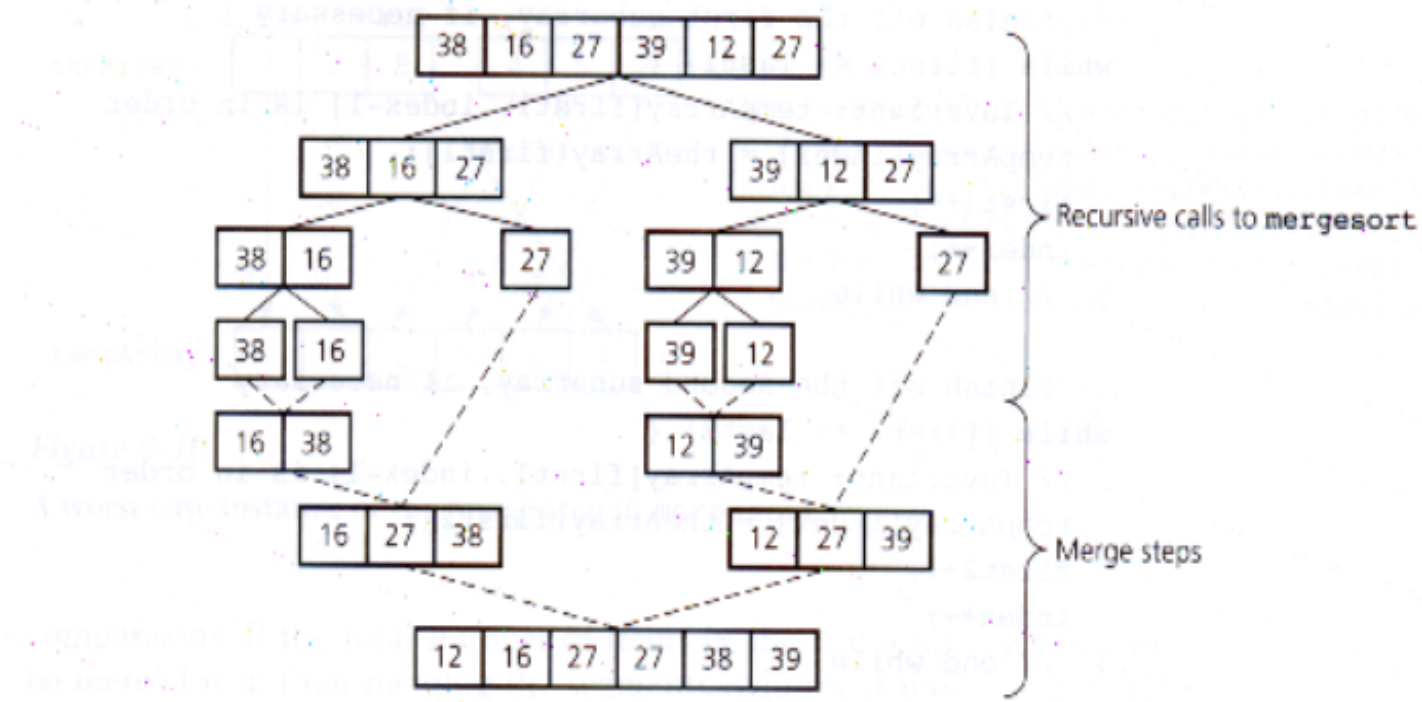
```
mergesort(theArray, first, last)
{     // Sorts theArray into ascending order.
      // Precondition: theArray[first..last] is an array.
      // Postcondition: theArray[first..last] is sorted in ascending order.
      //     1. Sort the first half of the array
      //     2. Sort the second half of the array
      //     3. Merge the two sorted halves.
//
 if (first < last)
{           // sort each half
      mid = (first + last)/2;   // get midpoint
            // sort left half of theArray[first..mid]
      mergesort(theArray, first, mid);
            // sort right half of theArray[mid+1..last]
      mergesort(theArray, mid+1, last);
            // merge the two halves
      merge(theArray, first, mid, last);
  }  // end if
}  // end mergesort
// if first >= last, then there is nothing to do
```

As we can see, most of the work in the merge sort is in the merge step.

How does this algorithm actually do the sort?

The recursive calls continue dividing the array into pieces until each piece contains only one item.

The algorithm then merges these smaller pieces into larger pieces until we eventually end up with one sorted array.

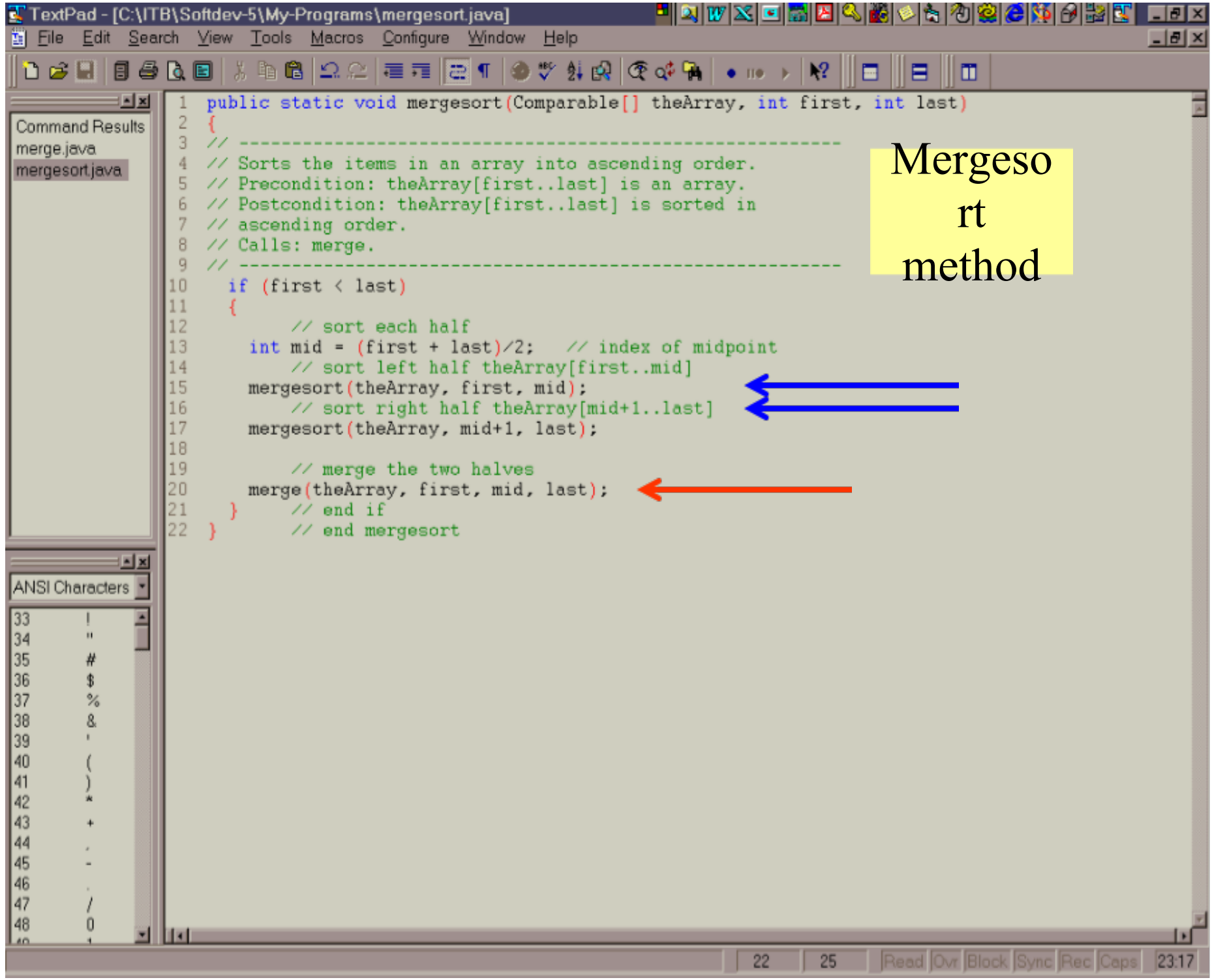


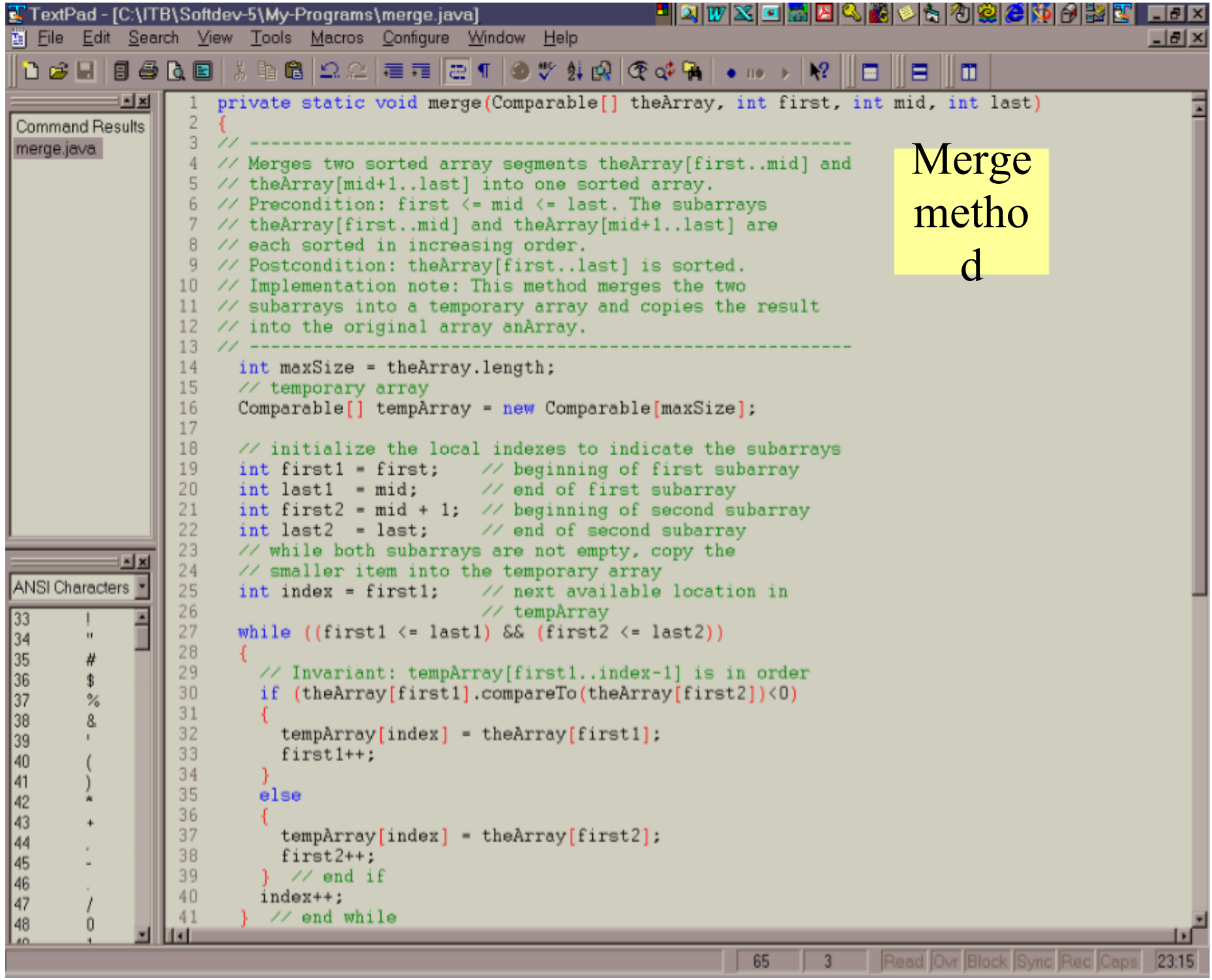
A mergesort of an array of six integers

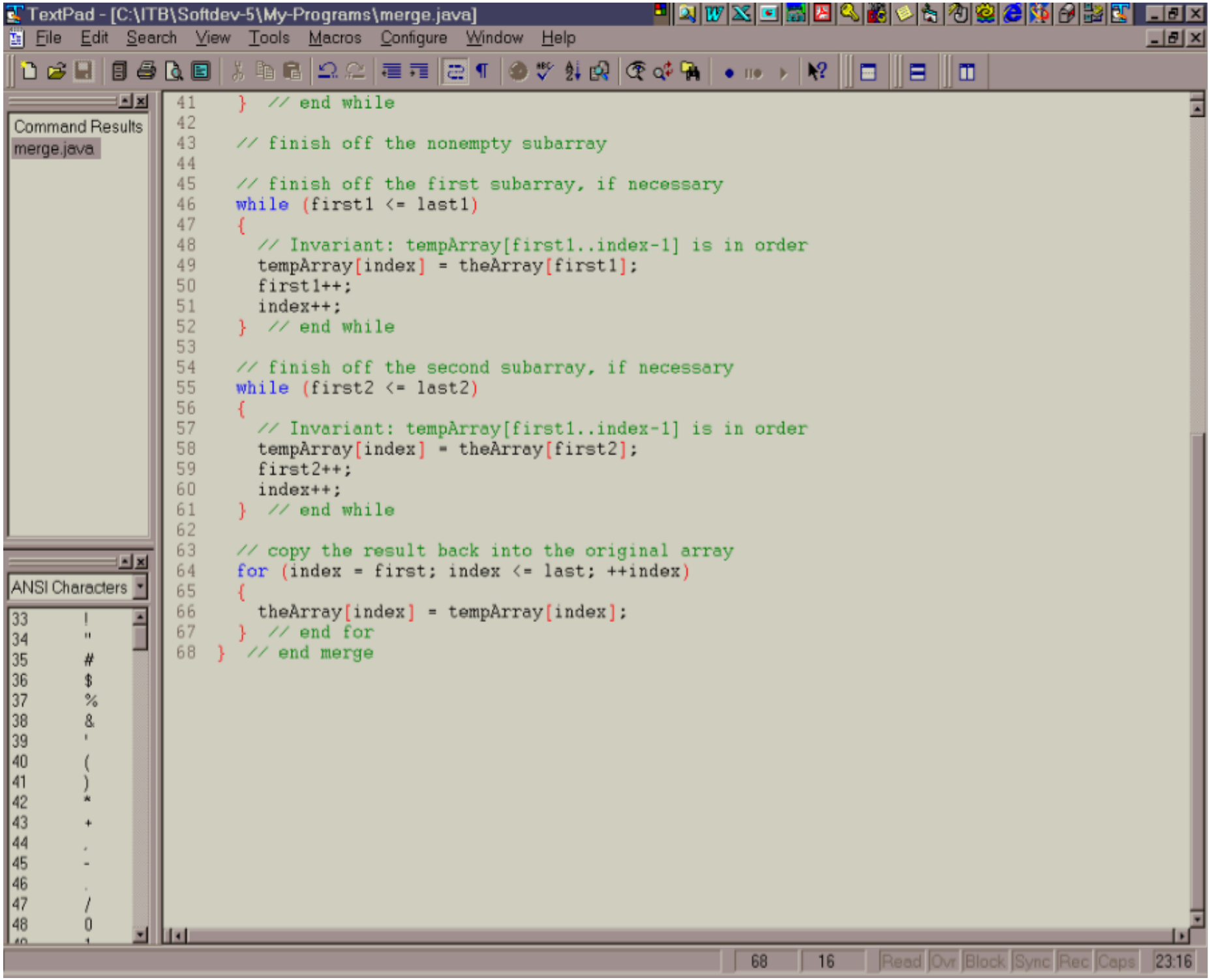
The following Java methods implement the mergesort algorithm.

To sort the arraytheArray of **n** items, ….

we would need to call the method mergesort by writing
mergesort(theArray, 0, n-1)


TextPad - [C:\ITB\Softdev-5\My-Programs\mergesort.java]
File Edit Search View Tools Macros Configure Window Help
Command Results
merge.java
mergesort.java
ANSI Characters
public static void mergesort(Comparable[] theArray, int first, int last)
{
// ----------
// Sorts the items in an array into ascending order.
// Precondition: theArray[first..last] is an array.
// Postcondition: theArray[first..last] is sorted in
// ascending order.
// Calls: merge.
// ----------
if (first < last)
{
// sort each half
int mid = (first + last)/2; // index of midpoint
// sort left half theArray[first..mid]
mergesort(theArray, first, mid);
// sort right half theArray[mid+1..last]
mergesort(theArray, mid+1, last);
// merge the two halves
merge(theArray, first, mid, last);
} // end if
} // end mergesort
Mergesort method
22 25 Read Ovr Block Sync Rec Caps 23:17

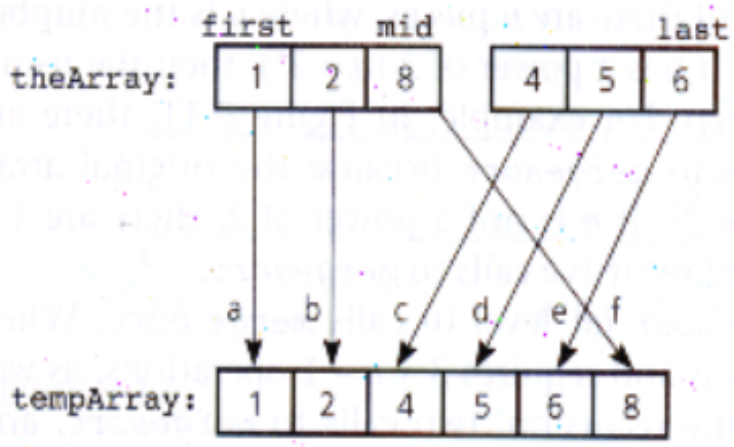

TextPad - [C:\ITB\Softdev-5\My-Programs\merge.java]
File Edit Search View Tools Macros Configure Window Help
Command Results
merge.java
ANSI Characters
33 !
34 "
35 #
36 $
37 %
38 &
39 '
40 (
41)
42 *
43 +
44 ,
45 -
46 .
47 /
48 0
private static void merge(Comparable[] theArray, int first, int mid, int last)
{
// --
// Merges two sorted array segments theArray[first..mid] and
// theArray[mid+1..last] into one sorted array.
// Precondition: first <= mid <= last. The subarrays
// theArray[first..mid] and theArray[mid+1..last] are
// each sorted in increasing order.
// Postcondition: theArray[first..last] is sorted.
// Implementation note: This method merges the two
// subarrays into a temporary array and copies the result
// into the original array anArray.
// --
int maxSize = theArray.length;
// temporary array
Comparable[] tempArray = new Comparable[maxSize];
// initialize the local indexes to indicate the subarrays
int first1 = first; // beginning of first subarray
int last1 = mid; // end of first subarray
int first2 = mid + 1; // beginning of second subarray
int last2 = last; // end of second subarray
// while both subarrays are not empty, copy the
// smaller item into the temporary array
int index = first1; // next available location in
// tempArray
while ((first1 <= last1) && (first2 <= last2))
{
// Invariant: tempArray[first1..index-1] is in order
if (theArray[first1].compareTo(theArray[first2])<0)
{
tempArray[index] = theArray[first1];
first1++;
}
else
{
tempArray[index] = theArray[first2];
first2++;
} // end if
index++;
} // end while
Merge method
65 3 Read Ovr Block Sync Rec Caps 23:15


TextPad - [C:\ITB\Softdev-5\My-Programs\merge.java]
File Edit Search View Tools Macros Configure Window Help
Command Results
merge.java
ANSI Characters
33 !
34 "
35 #
36 $
37 %
38 &
39 '
40 (
41)
42 *
43 +
44 ,
45 -
46 .
47 /
48 0
41 } // end while
42
43 // finish off the nonempty subarray
44
45 // finish off the first subarray, if necessary
46 while (first1 <= last1)
47 {
48 // Invariant: tempArray[first1..index-1] is in order
49 tempArray[index] = theArray[first1];
50 first1++;
51 index++;
52 } // end while
53
54 // finish off the second subarray, if necessary
55 while (first2 <= last2)
56 {
57 // Invariant: tempArray[first1..index-1] is in order
58 tempArray[index] = theArray[first2];
59 first2++;
60 index++;
61 } // end while
62
63 // copy the result back into the original array
64 for (index = first; index <= last; ++index)
65 {
66 theArray[index] = tempArray[index];
67 } // end for
68 } // end merge
68 16 Read Ovr Block Sync Rec Caps 23:16

Analysis of Mergesort

Because the **merge** step of the algorithm requires the most effort, we will start our analysis of that first….

Each merge step merges theArray[first..mid] and theArray[mid+1..last]



Merge the halves:

a. 1 < 4, so move 1 from `theArray[first..mid]` to `tempArray`
b. 2 < 4, so move 2 from `theArray[first..mid]` to `tempArray`
c. 8 > 4, so move 4 from `theArray[mid+1..last]` to `tempArray`
d. 8 > 5, so move 5 from `theArray[mid+1..last]` to `tempArray`
e. 8 > 6, so move 6 from `theArray[mid+1..last]` to `tempArray`
f. `theArray[mid+1..last]` is finished, so move 8 to `tempArray`

A worst-case instance of the merge step in *mergesort*

The diagram provides an example of a merge step that requires the maximum number of comparisons.

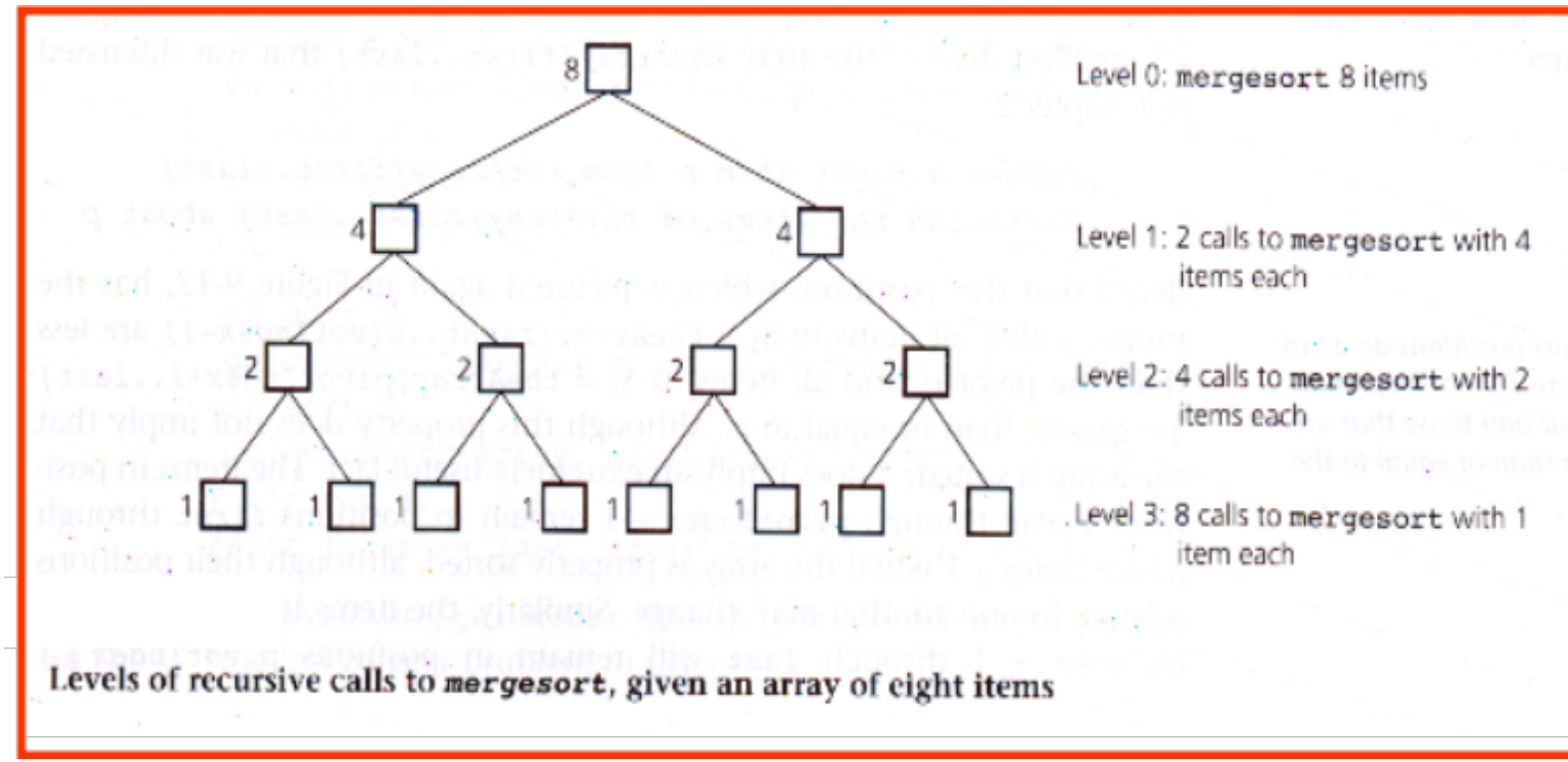
If the total number of items in the two array segments to be merged is **n**,
then merging the segments requires at most **n - 1** comparisons, i.e. 6 items will require 5 comparisons.

In addition, …

There are: **n** moves from the original array to the temporary array, and
n moves from the temporary array back to the original array.

Therefore, each merge step requires **3 * n – 1** major operations.

Each call to mergesort recursively calls itself twice.



Levels of recursive calls to `mergesort`, given an array of eight items

If the original call to mergesort is at level 0,
then two calls to mergesort occur at level 1 of the recursion.

Each of these calls then calls mergesort twice, so four calls to mergesort occur at level 2 of the recursion, and so on.

How many levels of recursion are there?

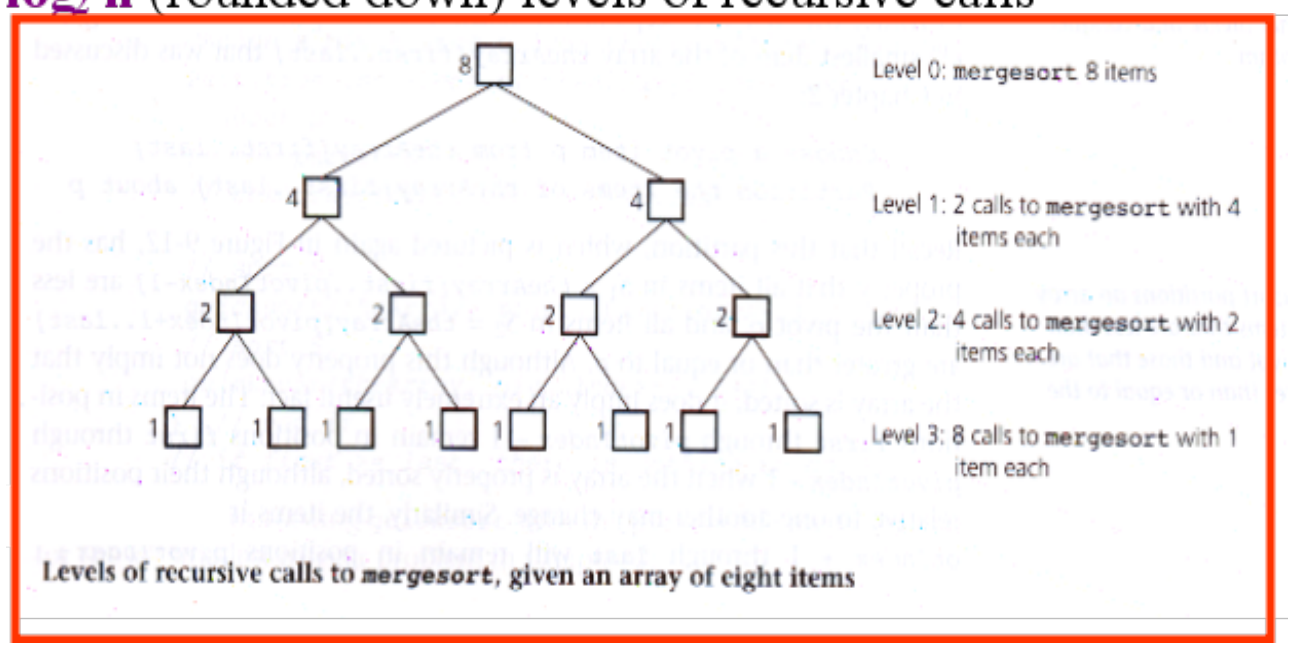
We can count them as follows….

The recursive calls continue until the array pieces each contain one item ., that is, until there are **n** pieces (where n is the number of items in the original array).

If **n** is a power of 2…… ($n = 2^k$),
Then the recursion goes $k = \log_2 n$ levels deep.

In the diagram there are three levels of recursive calls to mergesort because the original array contains 8 items, and **$8 = 2^3$.**

If **n** is *not* a power of two
Then there are **$1 + \log_2 n$** (rounded down) levels of recursive calls to mergesort



Levels of recursive calls to **mergesort**, given an array of eight items

The Worst and Average Cases of Mergesort

Each level of the recursion requires O(n) operations.

Because there are either $\log_2 n$ or $1 + \log_2 n$ levels, mergesort is $O(n * \log_2 n)$ in both the worst and average cases.

We know that $O(n * \log_2 n)$ is significantly faster than $O(n^2)$, the growth rate of the sorts we saw in an earlier lecture, the bubble sort, insertion sort and selection sort.

A comparison of Sorting Algorithms

The following table lists of a number of the major sorting algorithms.

We have studied bubble sort, selection sort, insertion sort, merge sort.

The other sorts are included to indicate how they compare to the sorts we have studied.

	Worst Case	Best Case
Selection sort	n^2	n^2
Bubble sort	n^2	n^2
Insertion sort	n^2	n^2
Merge sort	n * log n	n * log n
Quick sort	n^2	n * log n
Tree sort	n^2	n * log n
Heap sort	n * log n	n * log n

java.lang
Interface Comparable

All Known Implementing Classes:
Byte, Character, Double, File, Float, Long, ObjectStreamField, Short, String, Integer, BigInteger, BigDecimal, Date, CollationKey

public interface Comparable

This interface imposes a total ordering on the objects of each class that implements it. This ordering is referred to as the class's natural ordering, and the class's compareTo method is referred to as its natural comparison method.

Lists (and arrays) of objects that implement this interface can be sorted automatically by Collections.sort (and Arrays.sort). Objects that implement this interface can be used as keys in a sorted map or elements in a sorted set, without the need to specify a comparator.

A class's natural ordering is said to be consistent with equals if and only if (e1.compareTo((Object)e2)==0) has the same boolean value as e1.equals((Object)e2) for every e1 and e2 of class C.

It is strongly recommended (though not required) that natural orderings be consistent with equals. This is so because sorted sets (and sorted maps) without explicit comparators behave "strangely" when they are used with elements (or keys) whose natural ordering is inconsistent with equals. In particular, such a sorted set (or sorted map) violates the general contract for set (or map), which is defined in terms of the equals operation.

For example, if one adds two keys a and b such that (a.equals((Object)b) &&

a.compareTo((Object)b) != 0) to a sorted set that does not use an explicit comparator, the second add operation returns false (and the size of the sorted set does not increase) because a and b are equivalent from the sorted set's perspective.

Virtually all Java core classes that implement comparable have natural orderings that are consistent with equals. One exception is java.math.BigDecimal, whose natural ordering equates BigDecimals with equal values and different precisions (such as 4.0 and 4.00).

For the mathematically inclined, the relation that defines the natural ordering on a given class C is:

{(x, y) such that x.compareTo((Object)y) <= 0}.

The quotient for this total order is:

{(x, y) such that x.compareTo((Object)y) == 0}.

It follows immediately from the contract for compareTo that the quotient is an equivalence relation on C, and that the natural ordering is a total order on C. When we say that a class's natural ordering is consistent with equals, we mean that the quotient for the natural ordering is the equivalence relation defined by the class's equals(Object) method:

{(x, y) such that x.equals((Object)y)}.

Method Summary

int

compareTo(Object o)

Compares this object with the specified object for order.

Method Detail

compareTo

public int compareTo(Object o)

Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

The implementor must ensure sgn(x.compareTo(y)) == - sgn(y.compareTo(x)) for all x and y. (This implies that x.compareTo(y) must throw an exception iff y.compareTo(x) throws an exception.)

The implementor must also ensure that the relation is transitive: (x.compareTo(y)>0 &&y.compareTo(z)>0) implies x.compareTo(z)>0.

Finally, the implementer must ensure that x.compareTo(y)==0 implies that sgn(x.compareTo(z)) == sgn(y.compareTo(z)), for all z.

It is strongly recommended, but not strictly required that (x.compareTo(y)==0) == (x.equals(y)).

Generally speaking, any class that implements the Comparable interface and violates this condition should clearly indicate this fact. The recommended language is "Note: this class has a natural ordering that is inconsistent with equals."

Parameters: o - the Object to be compared.
Returns: a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.
Throws: ClassCastException - if the specified object's type prevents it from being compared to this Object.

Programs to do this week:

Write a Java program to implement the **mergesort**

In your program, populate an array (size 200) with objects that implement the Comparable interface. Each object should only have one instance variable (an integer) and two methods, int compareTo (that compares it's integer field with that of another parameter objects integer field) and getInt (that returns the integer value of the object that invoked it).

Print the array in a tabulated form in unsorted form,
in, for example, 20 columns x10 rows

Mergesort the array.

Print the array in a tabulated form in sorted form, as above.

Provide a heading to each display of the data.