

Sorting and Searching

CHAPTER

8

Critics search for ages for the wrong word, which, to give them credit, they eventually find.

—Peter Ustinov (1952)

Chapter Goals

- Java implementation of sorting algorithms
- Selection and insertion sorts
- Recursive sorts: mergesort and quicksort
- Sequential search and binary search

In each of the following sorting algorithms, assume that an array of n elements, $a[0]$, $a[1]$, ..., $a[n-1]$, is to be sorted in ascending order.

SORTS: SELECTION AND INSERTION SORTS

Selection Sort

This is a “search-and-swap” algorithm. Here’s how it works.

Find the smallest element in the array and exchange it with $a[0]$, the first element. Now find the smallest element in the subarray $a[1] \dots a[n-1]$ and swap it with $a[1]$, the second element in the array. Continue this process until just the last two elements remain to be sorted, $a[n-2]$ and $a[n-1]$. The smaller of these two elements is placed in $a[n-2]$; the larger, in $a[n-1]$; and the sort is complete.

Trace these steps with a small array of four elements. The unshaded part is the subarray still to be searched.

8	1	4	6	
1	8	4	6	after first pass
1	4	8	6	after second pass
1	4	6	8	after third pass

NOTE

1. For an array of n elements, the array is sorted after $n - 1$ passes.
2. After the k th pass, the first k elements are in their final sorted position.

Insertion Sort

Think of the first element in the array, $a[0]$, as being sorted with respect to itself. The array can now be thought of as consisting of two parts, a sorted list followed by an unsorted list. The idea of insertion sort is to move elements from the unsorted list to the sorted list one at a time; as each item is moved, it is inserted into its correct position in the sorted list. In order to place the new item, some elements may need to be moved down to create a slot.

Here is the array of four elements. In each case, the boxed element is “it,” the next element to be inserted into the sorted part of the list. The shaded area is the part of the list sorted so far.

8	1	4	6	
1	8	4	6	after first pass
1	4	8	6	after second pass
1	4	6	8	after third pass

NOTE

1. For an array of n elements, the array is sorted after $n - 1$ passes.
2. After the k th pass, $a[0], a[1], \dots, a[k]$ are sorted with respect to each other but not necessarily in their final sorted positions.
3. The worst case for insertion sort occurs if the array is initially sorted in reverse order, since this will lead to the maximum possible number of comparisons and moves.
4. The best case for insertion sort occurs if the array is already sorted in increasing order. In this case, each pass through the array will involve just one comparison, which will indicate that “it” is in its correct position with respect to the sorted list. Therefore, no elements will need to be moved.

Both insertion and selection sorts are inefficient for large n .

RECURSIVE SORTS: MERGESORT AND QUICKSORT

Selection and insertion sorts are inefficient for large n , requiring approximately n passes through a list of n elements. More efficient algorithms can be devised using a “divide-and-conquer” approach, which is used in both the sorting algorithms that follow.

Mergesort

Here is a recursive description of how mergesort works:

If there is more than one element in the array
 Break the array into two halves.
 Mergesort the left half.
 Mergesort the right half.
 Merge the two subarrays into a sorted array.

Mergesort uses a merge method to merge two sorted pieces of an array into a single sorted array. For example, suppose array $a[0] \dots a[n-1]$ is such that $a[0] \dots a[k]$ is sorted and $a[k+1] \dots a[n-1]$ is sorted, both parts in increasing order. Example:

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]
2	5	8	9	1	6

In this case, $a[0] \dots a[3]$ and $a[4] \dots a[5]$ are the two sorted pieces. The method call `merge(a, 0, 3, 5)` should produce the “merged” array:

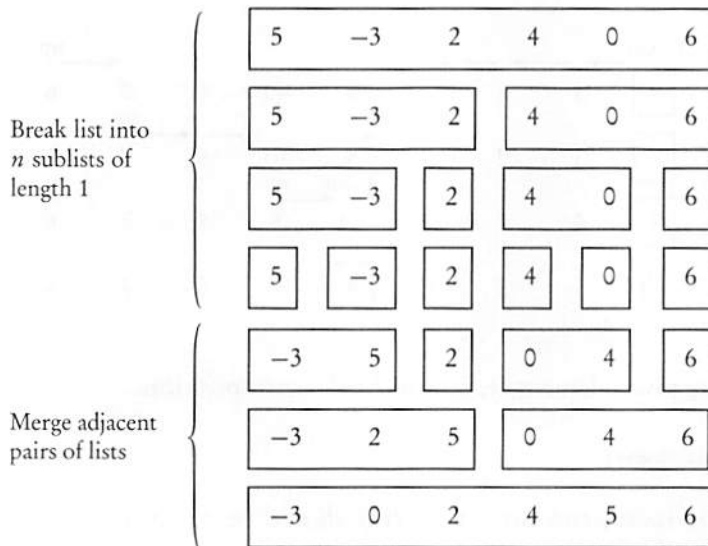
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]
1	2	5	6	8	9

The middle numerical parameter in `merge` (the 3 in this case) represents the index of the last element in the first “piece” of the array. The first and third numerical parameters are the lowest and highest index, respectively, of array a .

Here’s what happens in mergesort:

1. Start with an unsorted list of n elements.
2. The recursive calls break the list into n sublists, each of length 1. Note that these n arrays, each containing just one element, are sorted!
3. Recursively merge adjacent pairs of lists. There are then approximately $n/2$ lists of length 2; then, approximately $n/4$ lists of approximate length 4, and so on, until there is just one list of length n .

An example of mergesort follows:



The main disadvantage of mergesort is that it uses a temporary array.

Analysis of Mergesort:

1. The major disadvantage of mergesort is that it needs a temporary array that is as large as the original array to be sorted. This could be a problem if space is a factor.
2. Mergesort is not affected by the initial ordering of the elements. Thus, best, worst, and average cases have similar run times.

Quicksort

Optional topic

For large n , quicksort is, on average, the fastest known sorting algorithm. Here is a recursive description of how quicksort works:

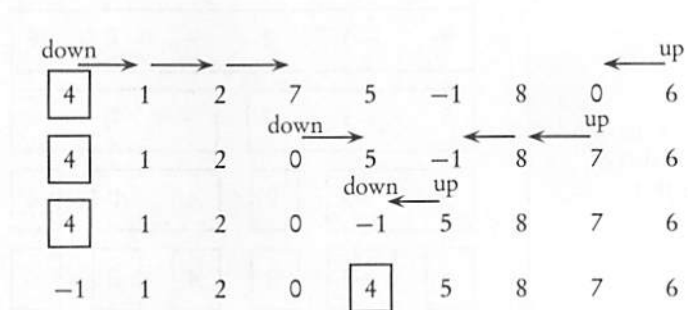
- If there are at least two elements in the array
- Partition the array.
 - Quicksort the left subarray.
 - Quicksort the right subarray.

The partition method splits the array into two subarrays as follows: a *pivot* element is chosen at random from the array (often just the first element) and placed so that all items to the left of the pivot are less than or equal to the pivot, whereas those to the right are greater than or equal to it.

For example, if the array is 4, 1, 2, 7, 5, -1, 8, 0, 6, and $a[0] = 4$ is the pivot, the partition method produces

-1 1 2 0 4 5 8 7 6

Here's how the partitioning works: Let $a[0]$, 4 in this case, be the pivot. Markers up and down are initialized to index values 0 and $n - 1$, as shown. Move the up marker until a value less than the pivot is found, or down equals up. Move the down marker until a value greater than the pivot is found, or down equals up. Swap $a[up]$ and $a[down]$. Continue the process until down equals up. This is the pivot position. Swap $a[0]$ and $a[pivotPosition]$.



Notice that the pivot element, 4, is in its final sorted position.

Analysis of Quicksort:

1. For the fastest run time, the array should be partitioned into two parts of roughly the same size.

The main disadvantage of quicksort is that its worst case behavior is very inefficient.

2. If the pivot happens to be the smallest or largest element in the array, the split is not much of a split—one of the subarrays is empty! If this happens repeatedly, quicksort degenerates into a slow, recursive version of selection sort and is very inefficient.
3. The worst case for quicksort occurs when the partitioning algorithm repeatedly divides the array into pieces of size 1 and $n - 1$. An example is when the array is initially sorted in either order and the first or last element is chosen as the pivot. Some algorithms avoid this situation by initially shuffling up the given array (!) or selecting the pivot by examining several elements of the array (such as first, middle, and last) and then taking the median.

(continued)

NOTE

For both quicksort and mergesort, when a subarray gets down to some small size m , it becomes faster to sort by straight insertion. The optimal value of m is machine-dependent, but it's approximately equal to 7.

SORTING ALGORITHMS IN JAVA

Unlike the container classes like `ArrayList`, whose elements must be objects, arrays can hold either objects or primitive types like `int` or `double`.

A common way of organizing code for sorting arrays is to create a sorter class with an array private instance variable. The class holds all the methods for a given type of sorting algorithm, and the constructor assigns the user's array to the private array variable.

Example

Selection sort for an array of `int`.

```
/* A class that sorts an array of ints from
 * largest to smallest using selection sort. */

public class SelectionSort
{
    private int[] a;

    //constructor
    public SelectionSort(int[] arr)
    { a = arr; }

    //Swap a[i] and a[j] in array a.
    private void swap(int i, int j)
    {
        int temp = a[i];
        a[i] = a[j];
        a[j] = temp;
    }
}
```

```

//Sort array a from largest to smallest using selection sort.
//Precondition: a is an array of ints.
public void selectionSort()
{
    int maxPos, max;

    for (int i = 0; i < a.length - 1; i++)
    {
        //find max element in a[i+1] to a[a.length-1]
        max = a[i];
        maxPos = i;
        for (int j = i + 1; j < a.length; j++)
            if (max < a[j])
            {
                max = a[j];
                maxPos = j;
            }
        swap(i, maxPos); //swap a[i] and a[maxPos]
    }
}

```

Only Comparable
objects can be sorted.

Note that in order to sort *objects*, the elements must be Comparable since you need to be able to compare them.

SEQUENTIAL SEARCH

Assume that you are searching for a key in a list of n elements. A sequential search starts at the first element and compares the key to each element in turn until the key is found or there are no more elements to examine in the list. If the list is sorted, in ascending order, say, stop searching as soon as the key is less than the current list element.

Analysis:

1. The best case has key in the first slot.
2. The worst case occurs if the key is in the last slot or not in the list. In the worst case, all n elements must be examined.
3. On average, there will be $n/2$ comparisons.

BINARY SEARCH

Binary search works
only if the array is
sorted on the search
key.

If the elements are in a *sorted* array, a divide-and-conquer approach provides a much more efficient searching algorithm. The following recursive pseudo-code algorithm shows how the *binary search* works.

Assume that $a[\text{low}] \dots a[\text{high}]$ is sorted in ascending order and that a method `binSearch` returns the index of key. If key is not in the array, it returns -1 .

```

if (low > high) //Base case. No elements left in array.
    return -1;
else
{
    mid = (low + high)/2;
    if (key is equal to a[mid]) //found the key
        return mid;
    else if (key is less than a[mid]) //key in left half of array
        < binSearch for key in a[low] to a[mid-1] >
    else //key in right half of array
        < binSearch for key in a[mid+1] to a[high] >
}

```

NOTE

When low and high cross, there are no more elements to examine, and key is not in the array.

Example: suppose 5 is the key to be found in the following array:

a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]
1	4	5.....7	9	12	15	20	21	

First pass: mid = (8+0)/2 = 4. Check a[4].
 Second pass: mid = (0+3)/2 = 1. Check a[1].
 Third pass: mid = (2+3)/2 = 2. Check a[2]. Yes! Key is found.

Analysis of Binary Search:

1. In the best case, the key is found on the first try (i.e., $(low + high)/2$ is the index of key.)
2. In the worst case, the key is not in the list or is at either end of a sublist. Here the n elements must be divided by 2 until there is just one element, and then that last element must be tested. An easy way to find the number of comparisons in the worst case is to round n up to the next power of 2 and take the exponent. For example, in the array above, $n = 9$. Suppose 21 were the key. Round 9 up to 16, which equals 2^4 . Thus you would need four comparisons to find it. Try it!

Chapter Summary

You should not memorize any sorting code. You must, however, be familiar with the mechanism used in each of the sorting algorithms. For example, you should be able to explain how the merge method of mergesort works, or what the purpose of the pivot element in quicksort is. You should know the best and worst case situations for each of the sorting algorithms.

Be familiar with the sequential and binary search algorithms. You should know that a binary search is more efficient than a sequential search, and that a binary search can only be used for an array that is sorted on the search key.