9

Sorting and Searching

Critics search for ages for the wrong word, which,
to give them credit, they eventually find.
—Peter Ustinov (1952)

Learning Objectives

In this chapter, you will learn:

- ➔ Sorts: selection and insertion sorts
- ➔ Recursive sorts: merge sort and quicksort
- ➔ Sorting algorithms in Java
- ➔ Sequential search
- ➔ Binary search

For 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]` ...`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

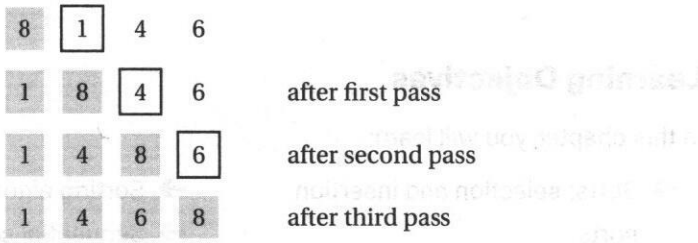
Note

1. For an array of n elements, the array is sorted after $n-1$ passes.
2. After the kth 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 to the right 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.



Note

1. For an array of n elements, the array is sorted after $n - 1$ passes.
2. After the kth pass, `a[0]`, `a[1]`, ..., `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: Merge Sort 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. Quicksort is not in the AP Java subset.

Merge Sort

Here is a recursive description of how merge sort works:

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

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

Merge sort uses a `merge` method to merge two sorted pieces of an array into a single sorted array. For example, suppose array `a[0] ... a[n-1]` is such that `a[0] ... a[k]` is sorted and `a[k+1] ... 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] ...a[3]` and `a[4] ...a[5]` are the two sorted pieces. The method call `merge(a,0,3,5)` should produce the "merged" array:

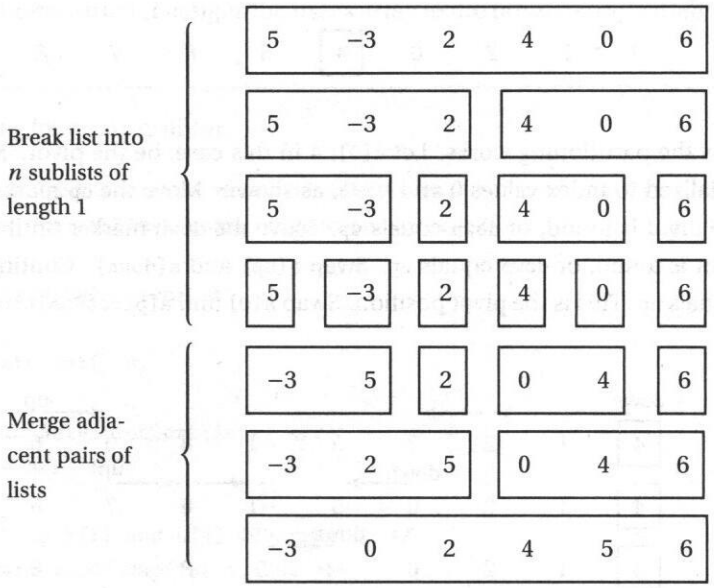
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]
1	2	5	6	8	9

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 indexes, respectively, of array `a`.

Here's what happens in merge sort:

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 merge sort follows:



Analysis of merge sort:

1. The major disadvantage of merge sort 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. Merge sort 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:

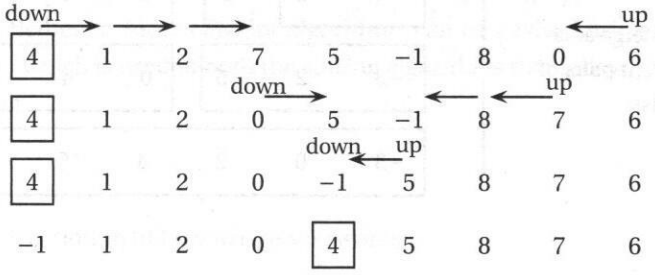
If there are at least two elements in the array,
    Partition the array.
    Quicksort the left subarray.
    Quicksort the right subarray.

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.

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.

OPTIONAL TOPIC

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

Note

For both quicksort and merge sort, 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;

    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]
        }
    }
}
```

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. (If the key is less than the current element, it will be less than all subsequent elements.)

Analysis:

1. The best case has the 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 pseudocode algorithm shows how the *binary search* works.

Assume that `a[low] ... a[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 that this algorithm can also be described iteratively. There are no recursive calls, just an adjustment of `mid` so that the algorithm searches to the left or the right.

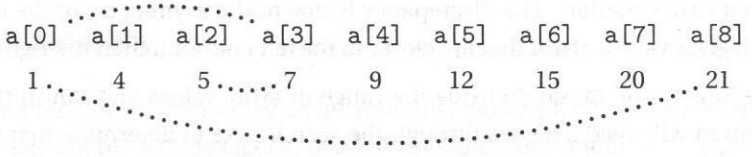
Again, assume that `a[low]...a[high]` is sorted in ascending order and that the method will return the index of `key`. If `key` is not in the array, the method will return −1.

```
while (low is less than or equal to high)
{
    int 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
        high = mid - 1;
    else  //key in right half of array
        low = mid + 1;
}
//If we get to here, then key is not in array.
return -1;
```

Note

1. After just one comparison, the binary search algorithm ignores one half of the array elements. This is true for both the iterative and recursive versions.
2. When `low` and `high` cross, there are no more elements to examine, and `key` is not in the array.

For example, suppose 5 is the key to be found in the following array:



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

Analysis of Binary Search

The number of comparisons for binary search in the worst case depends on whether n is a power of 2 or not.

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 array or is at an endpoint of the array. Here, the n elements must be divided by 2 until there is just one element, and then that last element must be compared with the key. 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 is 9. Suppose 21 were the key. Round 9 up to 16, which is 2^4. Thus you would need 4 comparisons with the key to find it. If n is an exact power of 2, the number of comparisons in the worst case equals the exponent plus one. For example, if the number of elements $n = 32 = 2^5$, then the number of comparisons in the worst case is $5 + 1 = 6$. Note that in this discussion, the number of comparisons refers to the number of passes through the search loop of the above algorithm—namely, the outer `else` piece of code.
3. There's an interesting wrinkle when discussing the worst case of a binary search that uses the above algorithm. The worst case (i.e., the maximum number of comparisons) will either have the key at an endpoint of the array, or be equal to a value that's not in the array. The opposite, however, is not necessarily true: If the key is at an endpoint, or a value not in the array, it is not necessarily a worst case situation.

As a simple example, consider the array `3, 7, 9, 11`, where `a[0]` is `3` and `a[3]` is `11`. The number of elements n equals 4, which is 2^2, an exact power of 2. The worst case for searching for a given key will be 3 comparisons, the exponent plus one.

- If the key is `11` (an endpoint of the array), the algorithm will need 3 passes through the search loop to find the key. This is a worst case. Here's how it works:
  1st pass: `low = 0 high = 3 mid = 1`
  2nd pass: `low = 2 high = 3 mid = 2`
  3rd pass: `low = 3 high = 3 mid = 3`
  The key is found during the 3rd pass when you examine `a[3]`. Thus a key of `11` represents a worst case.
- If the key is `3` (the other endpoint of the array), the algorithm will need 2 passes through the search loop to find the key. Here's how it works:
  1st pass: `low = 0 high = 3 mid = 1`
  2nd pass: `low = 0 high = 0 mid = 0`
  The key is found during the 2nd pass when you examine `a[0]`. Thus a key of `3` is not a worst case situation. The discrepancy is due to the asymmetry of the div operation, which gives values of `mid` that are closer to the left endpoint than the right.
- If the key is `1` or `20`, say (outside the range of array values and not in the array), the algorithm will need 3 passes through the search loop to determine that the key is not in the array, a worst case.
- If the key is `8`, say (not in the array but inside the range of array values), the algorithm will need just 2 passes through the search loop to determine that the key is not in the array. This is therefore not a worst case situation.
- If the key is `10`, say (not in the array but between `a[2]` and `a[3]` in this example), the algorithm will need 3 passes through the search loop to determine that the key is not in the array, a worst case! Here is how it works:

1st pass: `low = 0 high = 3 mid = 1`
2nd pass: `low = 2 high = 3 mid = 2`
3rd pass: `low = 3 high = 3 mid = 3`
When `a[3]` is found to be greater than `key`, the value of `low` becomes 4, while `high` is still 3, which means that the test `if (low > high)` becomes true and is a base case that terminates the algorithm. There are no further comparisons with `key`.

Here is another example, where n is not a power of 2.

Suppose the array is 1, 3, 5, 7, 9. Here n is 5. To find the number of passes in the worst case, round up to the nearest power of 2, which is 8 or 2^3. In the worst case, the number of passes through the search loop will be 3:

- If the key is 1, there will be 2 passes to find it, which is not a worst case.
- If the key is 9, there will be 3 passes to find it, which is a worst case.
- If the key is 8, there will be 3 passes to find it, which is a worst case.
- If the key is 4, there will be 2 passes to find it, which is not a worst case.
- If the key is any value outside the range of 1 – 9, there will be 3 passes to find it, which is a worst case.

The lessons from these examples is that not every key that is not in the array represents a worst case.

Here are some general rules for calculating the maximum number of loop passes in different binary search situations. In each case it's assumed that the algorithm given in this book is used.

- If n, the number of elements, is not a power of 2, round n up to the nearest power of 2. The number of passes in the worst case equals the exponent.
- If n is a power of 2, the number of passes in the worst case equals the exponent plus one.
- Irrespective of n, the worst case will always involve a key that is either at the right endpoint or not in the array.
- Irrespective of n, any key that is not in the array and is less than `a[0]` or greater than `a[n-1]` will be a worst case situation.
- Irrespective of n, any key that is between `a[0]` and `a[n-1]` but is not in the array may or may not be a worst case situation.

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 merge sort works, or how many elements are in their final sorted position after a certain number of passes through the selection sort loop. 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.