Chapter 8

Sorting and Searching

One of the essential operations over a data structure is *searching* for a particular value. For example, we can find such a value in an array by checking every element of the array in-order.

```java
/**
 * Performs linear search to find a value in an array of ints.
 * @param arr an array of integers
 * @param v the target value
 * @return the index of the first occurrence of v in the array or -1
 *         if v is not in the array
 */
public static int indexOf(int[] arr, int v) {
    for (int i = 0; i < arr.length; i++) {
        if (arr[i] == v) { return i; }
    }
    return -1;
}
```

This algorithm is appropriately named *linear search* and has worst-case $\mathcal{O}(N)$ time complexity where N is the length of the input array corresponding to checking every element in the array.

While linear search takes advantage of the sequential nature of an array (and in general, any sequential data structure), it does not take *full* advantage of this property. In particular, if the array is *ordered*, then we can employ *binary search* to avoid searching every element of the array.

```java
/**
 * Performs binary search to find a value in a sorted array of ints.
 * @param arr a sorted array of integers
 * @param v the target value
 * @return the index of the first occurrence of v in the array or -1
 *         if v is not in the array
 */
public static int bsearch(int[] arr, int v) {
    return bsearch(arr, v, 0, arr.length);
}
```

```java
/**
 * Performs binary search to find a value in a subregion of a sorted array of ints.
 * @param arr a sorted array of integers
 * @param v the target value
 * @param lo the lower bound of the subarray (inclusive)
 * @param hi the upper bound of the subarray (exclusive)
 * @return the index of the first occurrence of v in the array or -1
 *         if v is not in the array
 */
public static int bsearch(int[] arr, int v, int lo, int hi) {
    if (lo > hi) {
        return -1;
    } else {
        mid = lo + (hi - lo) / 2;
        if (v < arr[mid]) {
            return bsearch(arr, v, lo, mid);
        } else if (v > arr[mid]) {
          return bsearch(arr, v, mid + 1, hi);
        } else {
            return mid;
        }
    }
}
```

As we discussed earlier, binary search has $\mathcal{O}(N \log N)$ worst-case time complexity by repeatedly chopping the array of the array under consideration in half. The downside, of course, is that the input array must be sorted. However, since searching is such a fundamental operation over a data structure, it is frequently more efficient to sort our data before we analyze it.

In this chapter, we investigate fundamental sorting algorithms over sequential data structures. There are many such sorting algorithms with unique trade-offs between each other in terms of performance and behavior.

8.1 Quadratic Sorts

One natural way we can deriving sorting algorithms is to employ *invariant design* directly on the traversal of an array. It makes sense to attempt to traverse a list left-to-right to try to place each element in its sorted position. Depending on the invariant we place on this traversal, we obtain different sorting algorithms.

For example, consider the following invariant:

Invariant 1. *When we are at position i in the array, the i smallest elements are to the left of the array.*

Pictorally, we represent this invariant in an array as follows:

```
[/////////////|           ]
  smallest    ^
    elts      i
   sorted
```

To implement this sorting algorithm, we need to implement a loop that satisfies the invariant:

```
for (int i = 0; i < arr.length; i++) {
    // Ensure that the i smallest elements of the array
    // are in the indices 0, ..., i
}
```

This requires that we find the smallest element in unsorted region of the array and then place it at position i—the smallest element in the unsorted region becomes the largest element in the sorted region.

```
public static void swap(int[] arr, int i, int j) {
    int temp = arr[i];
    arr[i] = arr[j];
    arr[j] = temp;
}

public static void selectionSort(int[] arr) {
    for (int i = 0; i < arr.length - 1; i++) {
        int lowestIndex = i;
        for (int j = i; j < arr.length; j++) {
            if (arr[j] < arr[lowestIndex]) {
                lowestIndex = j;
            }
        }
        swap(i, lowestIndex);
    }
}
```

Note that we do not need to consider the final element in the outer loop because we know by our invariant that it must be the largest element in the array by virtue of being the only one in the unsorted region. This sorting algorithm is called *selection sort*.

For example, consider the following array $[3, 1, 5, 9, 4]$. Here is an example step-by-step execution of selection sort on the array:

$$[\underline{3}, 1, 5, 9, 4]$$
$$[1, \underline{3}, 5, 9, 4]$$
$$[1, 3, \underline{5}, 9, 4]$$
$$[1, 3, 4, \underline{9}, 5]$$
$$[1, 3, 4, 9, 5]$$

The underline value represents i before the execution of each inner loop of the algorithm.

To analyze the runtime of this sorting algorithm, we note that the first iteration of the outer loop runs for n iterations, the second iteration $n - 1$ times, the third iteration $n - 2$, and so forth. Inside each inner loop, we perform a constant number of array accesses. Afterwards, we perform an array swap. Thus, the time complexity runtime is $\mathcal{O}(n^2)$. Note that this is irrespective of the shape of the array so it is the best-, worst-, and average-case runtimes.

Can we do better? What if we *relaxed* the invariant so that rather than demanding that the sorted region contain the smallest elements of the array, it simply contains sorted elements. By doing so, we still obtain a sorted array in the end, but also gain some flexibility in our algorithm that we can use to make it more efficient in some cases.

Let's try working with the following invariant:

Invariant 2. *When we are at position i in the array, the elements up to i are in sorted order.*

Pictorally, the invariant looks as follows:

```
[///////////|           ]
   sorted   ^
            i
```

Intuitively, by working with the element at i rather than seeking the next smallest element in the unsorted portion of the array, we cut down on the amount of array accesses we make. For example, consider sorting the same array $[3, 1, 5, 9, 4]$ using this technique:

$$[3, \underline{1}, 5, 9, 4]$$
$$[1, 3, \underline{5}, 9, 4]$$
$$[1, 3, 5, \underline{9}, 4]$$
$$[1, 3, 5, 9, \underline{4}]$$
$$[1, 3, 5, \underline{4}, 9]$$
$$[1, 3, 4, 5, 9]$$

For each element we encounter (underlined above), we *push* it into its proper position in the sorted region of the array. We accomplish this by swapping that element with the element to its left whenever they are out of order. For example, at the first step of the algorithm, we swap 3 and 1 because they are out of order. Likewise, to put 4 in its proper position, we swap it several times to get into the 2nd index of the array.

With this, we can implement this sort, called *insertion sort*, as follows:

```
public static void insertionSort(int[] arr) {
    for (int i = 1; i < arr.length; i++) {
        for (int j = i; j > 0 && arr[j-1] > arr[j]; j--) {
            swap(arr, j, j-1);
        }
    }
}
```

Rather than seeking for the smallest element in the unsorted region of the array, the inner for-loop swaps element i downward into the sorted region until it is in its sorted position.

Like selection sort, insertion sort exhibits a doubly-nested for-loop over the lenghth of the input array which, at first glance, implies that its runtime complexity is $\mathcal{O}(n)$. However, the behavior of the inner for-loop depends on the shape of the array. For example, if the array is already sorted, then every iteration of the inner for-loop immediately terminates because for any j, `arr[j-1] < arr[j]`. This means that in this best case, the runtime of insertion sort is $\mathcal{O}(n)$. Sorting an already-sorted array may feel like an unlikely corner case. However, in practice this happens with some frequency because sometimes our data already comes sorted and it already costs $\mathcal{O}(n)$ time to check that this is the case.

In contrast, the worst case is when the array is in *reverse sorted* order. When this is true, every j encountered in the inner-for loop is the smallest element to be inserted into the sorted region of the array. Thus, every run of the inner-for loop performs $j - 1$ swaps to get the jth element to the front of the array. Because j ranges from 1 to `arr.length`, this means that in the worst case, insertion sort still performs in $\mathcal{O}(n^2)$ time. In practice, insertion sort performs swaps at each run of the inner-loop proportional to the length of the array, so the average case runtime is also $\mathcal{O}(n^2)$.

8.2 Merge Sort

There are many other quadratic sorts to consider. However, they all have the same $\mathcal{O}(n^2)$ performance, and most have worse properties than insertion sort, which will be our go-to quadratic sort. Can we do better than $\mathcal{O}(n^2)$? It turns out we can by employing the same *divide-and-conquer* technique that we employed to implement binary search.

Suppose that we are able to sort two halves of an array.

```
[////////////][\\\\\\\\\\\\]
  sorted        sorted
```

Can we combine them into a sorted whole? We can do this through a process called *merging*. Consider a concrete array $[1, 4, 9, 2, 3, 8]$ where the first three elements are sorted and the last three elements are sorted. To merge these two sub-arrays into a sorted whole, we first create an auxiliary array the same size as the input array that holds our sorted result. We then start with two pointers into the array, each at the start of the two sub-arrays.

$$[\underline{1}, 4, 9, \underline{2}, 3, 8] \longrightarrow [\underline{0}, 0, 0, 0, 0, 0]$$

We maintain the invariants that:

(a) The two pointers in the input array point to the smallest elements in each of the sub-arrays that have not yet been sorted.

(b) The elements to the left of the pointer in the auxiliary array are the smallest elements of the input array in sorted order.

At the end of the merge operation, the auxiliary array contains all of the elements of the input array in sorted order. We can then copy these elements back into the input array as our final result.

To do this, at every step of the merge operation, we take the minimum of the two elements pointed at in our input array and then copy it into the current position of the auxiliary array. We then advance the pointer that pointed to the element as well as the auxiliary array's pointer. Here's an example run of the merge operation on the array above:

$$\begin{aligned}
[\underline{1}, 4, 9, \underline{2}, 3, 8] &\longrightarrow [\underline{0}, 0, 0, 0, 0, 0] \\
[1, \underline{4}, 9, \underline{2}, 3, 8] &\longrightarrow [1, \underline{0}, 0, 0, 0, 0] \\
[1, \underline{4}, 9, 2, \underline{3}, 8] &\longrightarrow [1, 2, \underline{0}, 0, 0, 0] \\
[1, \underline{4}, 9, 2, 3, \underline{8}] &\longrightarrow [1, 2, 3, \underline{0}, 0, 0] \\
[1, 4, \underline{9}, 2, 3, \underline{8}] &\longrightarrow [1, 2, 3, 4, \underline{0}, 0] \\
[1, 4, \underline{9}, 2, 3, 8] &\longrightarrow [1, 2, 3, 4, 8, \underline{0}] \\
[1, 4, 9, 2, 3, 8] &\longrightarrow [1, 2, 3, 4, 8, 9]
\end{aligned}$$

Note that we stop comparing sides once we reach the end of one of the sub-arrays. In the case above, we reached the end of the right-hand sub-array first. The remaining elements of the left-hand sub-array are then copied into the auxiliary array—they are the largest elements of the overall array, already in order.

With the merge operation defined, we can define *merge sort* over a sub-array of the overall input array as follows:

- If the sub-array to sort is size 1 or less, then it is already trivally sorted.
- Otherwise, recursively sort the left- and right-hand halves of the sub-array and then merge the two into a final sorted whole.

To analyze the time complexity of merge sort, we first analyze the runtime of the merge operation. Given a sub-array of length n, the merge operation touches every element of that sub-array twice—once to copy it into the auxiliary array and once to copy it back in sorted order after-the-fact. This happens irrespective of the composition of the sub-array, so the merge operation performs $2n$ operations.

Now, let's consider the runtime of the complete sorting algorithm. It is recursive, and thus requires a recurrence relation $T(n)$ describing the runtime of the algorithm over the length of the array n. The base case is when the array has at most one element where we immediately return. In the recursive case, we recursively sort both halves of the input array and perform merge in $2n$ steps. This yields the following recurrence:

$$\begin{aligned} T(1) &= 0 \\ T(n) &= 2n + 2T(\frac{n}{2}) \end{aligned}$$

We can derive a closed-form solution for this recurrence by rewriting the formula in terms of k unfolding of the recurrence and setting k equal to the case when the recurrence bottoms out. After k unfoldings, the recurrence yields:

$$\begin{aligned} T(n) &= 2n + 2T(\frac{n}{2}) \\ &= 2n + 2(2\frac{n}{2} + 2T(\frac{n}{4}) \\ &= 2n + 2n + 4T(\frac{n}{4}) \\ &= 2n + 2n + 2(4\frac{n}{4} + 2T(\frac{n}{8})) \\ &= 2n + 2n + 2n + 2T(\frac{n}{8}) \\ &= k2n + 2^k T(\frac{n}{2^k}) \end{aligned}$$

The recurrence bottoms-out when $\frac{n}{2^k} = 1$. Solving in terms of n, this occurs when $k = \log_2 n$. Plugging back into the recurrence yields:

$$\begin{aligned} T(n) &= 2n \log_2 n + 2^{\log_2 n} T(\frac{n}{2^{\log_2 n}}) \\ &= 2n \log_2 n \end{aligned}$$

Thus, $T \in \mathcal{O}(n \log n)$.

Similarly, let's consider the space complexity of merge sort. Every call to merge allocates a separate array the same size as the input. This yields the following recurrence for the space complexity:

$$S(1) = 0$$
$$S(n) = n + 2S(\frac{n}{2})$$

Performing a similar analysis above yields that $S \in \mathcal{O}(n \log n)$. However, we can do better than this by simply passing in a single array the size of the overall input array to merge sort once and perform all our scratch work in that array. This requires a single allocation of a n-length array upfront which makes the space complexity $S \in \mathcal{O}(n)$.

8.3 Quicksort

Can we do better than merge sort? The runtime is nice, but can we get rid of that $\mathcal{O}(n)$ space complexity? To do this, let's keep the "divide-and-conquer" style of behavior (as this is what nets us that nice linearithmic time complexity), but vary other aspects of the algorithm. Summarizing merge sort's behavior:

1. Find the midpoint of the array.
2. Recursively sort the two halves of the array.
3. Merge the two sorted halves into a completely sorted array.

Note that we recursively sort and then merge. What if we flipped this around, performing the actual "sorting" behavior first, and then recursively sorted the halves? This idea forms the basis of the final sorting algorithm that we will cover: *Quicksort*.

Let's first imagining choosing the midpoint of the array. We cannot merge the two halves of the array at this point because they are unsorted. But what we can we do to make progress? We can *partition* the array based off of the value at the midpoint of the array. If our array has this shape beforehand:

```
[      ][ ][      ]
        mid
```

We want to modify it so that it has the following shape:

```
[ <mid ][ ][ mid> ]
        mid
```

That is, all the elements less than `mid` are to the left and all the elements greater than `mid` are to the right. This way, (1) we know `mid` is in the correct position and (2) we can recursively sort the left- and right-hands of the array independently. Note that the left- and right-hand partitions of the array may not be sorted yet—the goal of our recursive calls are to sort these halves.

How do we perform this partition operation? Let's imagine how we might do this on the following array:

$$[3, 9, 2, 8, 6, 4, 1, 7, 5]$$

The midpoint is the element 6. How do we partition this array? We can *swap* elements to the left of the midpoint with an element to the right of the midpoint if they are not in the correct position; for example, 1

and 9 ought to be swapped with respect to the pivot, 6. However, by doing so, the resulting left-hand and right-hand sides may not be equal in size. For example, if we partition the following array:

$$[1, 1, 7, 1, 6, 1, 1, 1, 1]$$

Then the result should be:

$$[1, 1, 1, 1, 1, 1, 1, 6, 7]$$

The left-hand partition is far larger than than the right-hand partition because only one element in the array is greater than 6.

Therefore, we cannot simply leave the midpoint where it is as it might move as a result of the partition. To alleviate this, we first move the midpoint to the end of the array via a swap:

$$[3, 9, 2, 8, 5, 4, 1, 7, 6]$$

And then we proceed to swap elements with our two fingered approach similar to the merge operation, except that left-hand finger starts on index 0 and the right-hand finger starts on the second-to-last element. Each finger then searches for the next element on its respective side to swap, and then we swap the elements. The procedure ends when our fingers cross. Here's an example execution of the procedure using our original array:

$$[\underline{3}, 9, 2, 8, 5, 4, 1, \underline{7}, 6]$$
$$[3, \underline{9}, 2, 8, 5, 4, 1, \underline{7}, 6]$$
$$[3, \underline{9}, 2, 8, 5, 4, \underline{1}, 7, 6]$$
$$[3, \underline{1}, 2, 8, 5, 4, \underline{9}, 7, 6]$$
$$[3, 1, 2, \underline{8}, 5, 4, \underline{9}, 7, 6]$$
$$[3, 1, 2, \underline{8}, 5, \underline{4}, 9, 7, 6]$$
$$[3, 1, 2, \underline{4}, 5, \underline{8}, 9, 7, 6]$$
$$[3, 1, 2, 4, \underline{5}, \underline{8}, 9, 7, 6]$$
$$[3, 1, 2, 4, 5, \underline{8}, 9, 7, 6]$$

After this swapping process, we know that everything to the left our fingers is less than the midpoint and everything to the right of our fingers is greater than the midpoint. We can then swap the midpoint back in where our pointers crossed as our pointers now point to an element larger than the midpoint.

```
[3, 1, 2, 4, 5, 6, 9, 7, 8]
 ////////////     \\\\\\\
      <6             >6
```

Note that the fact that 6 was the midpoint of the array did not matter in the end. Indeed, we could have chosen any element to serve as the *pivot* of the partition operation. And as you might guess, our choice of pivot heavily influences the runtime of quicksort. In particular, we want to choose pivots such that we partition in the array to obtain subarrays of size $n/2$. Poor choices of pivots result in dividing up the array so that one array has size 1 and the other than size $N - 1$.

Given that we have some way to choose a good pivot, quicksort proceeds as follows:

1. Choose a pivot in the array.

2. *Partition* the array into two subarrays using the pivot.
3. Recursively quicksort the two subarrays.

As noted above, our choice of pivot heavily influences the complexity of quicksort. In the best case, our choices of pivots allows us to evenly partition the array in half. From our description of the partition operation above, we see that we walk the array with our two fingers, so it takes linear time. Overall, this means that the recurrence for quicksort in the best case is $T(n) = n + 2T(n/2)$ which we know from mergesort means that the algorithm takes $\mathcal{O}(n \log n)$ time in the best case.

In the worst case, the pivot divides up the array so that we have a subarray of size 1 and a subarray of size $n - 1$. The recurrence here is now $T(n) = n + T(n-1) + T(1)$. The base case of quicksort, like mergesort, is trivial so $T(1)$ is just 1. Expanding the recurrence for a pattern yields:

$$\begin{aligned} T(n) &= n + 1 + T(n-1) \\ &= n + 1 + ((n-1) + T(n-1) \\ &= 2n + 1 + T(n-2) \\ &= 3n + T(n-3) \end{aligned}$$

After k expansions, we have $T(n) = kn + 1 - k + T(n-k)$. This process bottoms out when $n = k$ where we have $T(n) = n^2 + 1 - n + 1 = n^2 - n + 2$. The quadratic term dominates so we have $T \in \mathcal{O}(n^2)$.

On average, we divide up the array by some fraction—not necessarily $n/2$ but say, $n/4$ and $3n/4$ which results in a linearithmic ($\mathcal{O}(n \log n)$) runtime.

This seems worse than mergesort which has a consistent $\mathcal{O}(n \log n)$ runtime. However, unlike mergesort, quicksort takes $\mathcal{O}(\log n)$ space due to recursive function calls; it does not allocate extra anything on the heap.

So how do we choose the pivot? Ideally, we would like to choose the median of the array which guarantees that we partition the array optimally. However, because the array is unsorted, finding the median takes $\mathcal{O}(n^2)$ time which would ruin the runtime of the sort. Thus, we need a pivot-choosing method that is (1) simple, (2) approximates the best-case scenario of the median, and (3) avoids the worst-case scenario of choosing the smallest or largest element.

A simple choice is the first element of the array. However, if the array is already sorted, then this element is the smallest, leading to the worst-case behavior described above. Another choice is to simply choose a random element. This makes it unlikely to fall into the worst case scenario, but the behavior of mergesort then becomes more unpredictable.

Our original choose of the midpoint of the array is actually somewhat sensible. In the case when the array is sorted, we choose the optimal value. However, this choice does not approximate the median value at all when the array is unsorted.

A better choice that avoids the worst case and approximates the median is the *median-of-three* rule. The median of three rule says the following:

> Choose the pivot to be the median of the values found in the first, middle, and last positions of the array.

Note that we can perform this pivot choice operation in constant time as it is just a small number of comparisons between the first, middle, and last values. This is strictly better than simply choosing the midpoint—in the case where the array is sorted, we correctly choose the middle value. In the case where the array is unsorted, we at least get the median of some range of numbers. We might get unlucky and choose the three smallest numbers, but at least we are *guaranteed* to never fall into the absolute worst case scenario of choosing the smallest or largest values of the array.

8.4 Generic Constraints

So far, you have likely written your sorting routines in terms of integer arrays. However, we would like to sort arrays of any type without having to duplicate code. To do this in Java, we use generics:

```
public static <T> void insertionSort(T[] arr) {
    for (int i = 0; i < arr.length; i++) {
        for (int j = i; j < arr.length; j++) {
            if (arr[j] < arr[i]) {
                T temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }
    }
}
```

Like classes, we can declare a static method as generic by introducing a type parameter enclosed in angle brackets. For a method, the type parameter declaration goes before the return type (above, it is `<T>`). Alternatively, we can also enclose the method in a generic class and use the type parameter from the class declaration. (Note that instance methods cannot be generic—their enclosing class must be declared generic instead.)

However, this method as-is does not compile. Java gives the following error for the above code snippet:

```
Sort.java:5: error: bad operand types for binary operator '<'
                if (arr[j] < arr[i]) {
                           ^
  first type:  T
  second type: T
  where T is a type-variable:
    T extends Object declared in method <T>insertionSort(T[])
1 error
```

The problem stems from the fact that `T` is a generic type. The only operations we can perform on a `T` are those guaranteed by the `Object` class (since `Object` is the root of the Java class hierarchy). Comparison (*i.e.*, the less-than operator) is not one of these universal operations. And indeed, some types may not have the less-than operator defined for them.

We need some mechanism for *constraining* instances of this generic type to be only those that have some suitable definition of comparison. We do this through *generic type constraints*.

```
public static <T extends Comparable<T>> void insertionSort(T[] arr) {
    for (int i = 0; i < arr.length; i++) {
        for (int j = i; j < arr.length; j++) {
            if (arr[j].compareTo(arr[i]) < 0) {
                T temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
```

```
        }
    }
}
```

Here, we augment the type parameter declaration to be a type parameter called `T` that *must implement the `Comparable<T>` interface*. `Comparable<T>` is an interface in the Java standard library that objects implement to define what it means to compare them to other objects of the type parameter. Here it the complete definition of the interface:

```
public interface Comparable<T> {
    public int compareTo(T o);
}
```

Implementors provide a method, `compareTo`, that takes in an object of the given generic type and produces the following return value:

- A negative integer if this object is "less than" the other object.
- 0 if this object is "equal" to the other object.
- A positive integer if this object is "greater than" the other object.

Note how the `compareTo` method supercedes the funcitonality of the less-than, greater-than, and equals operators. So rather than performing a less-than check in insertion sort, we use `compareTo` to determine if the given element is the new minimum.

To use `insertionSort` now, we must keep in mind that it requires an array of reference type. This means we can no longer pass in an `int` array; we must pass in an `Integer` array instead:

```
// NOTE: this works because Java auto-boxes the ints into Integer objects...
Integer[] arr = new Integer[] { 4, 6, 1, 2, 9, 9, 10, 4, 2 };
insertionSort(arr);
```

The `Comparable` interface allows us to state what it means to compare any two types of objects. In other words, it allows us to impose an *ordering* on these types of objects. Most commonly, we're interested in specifying an ordering on objects of the *same type*, for example ordering dogs by age:

```
public class Dog implements Comparable<Dog> {
    private int age;
    // ...
    @Override
    public int compareTo(Dog other) {
        if (this.age < other.age) {
            return -1;
        } else if (this.age > other.age) {
            return 1;
        } else {
            return 0;
        }
    }
}
```

The Java standard library defines what it means for all the wrapper classes to be compared to each other (*e.g.*, `Integer`, `Double`) as well as common classes such as `String`. But we are free to define what the ordering is between types and also define what combinations of types have an ordering.

8.4.1 Wildcards and Type Bounds*

Our `insertionSort` method is close to being perfect, but not quite satisfactory. The generic type constraint requires that the object provide an implementation for being `Comparable` to objects of its own type. But we ought to be more flexible than this. Note that we use the `Comparable` interface to compare elements of the same type. If our object provides an implementation of `Comparable<T>` where `T` is a *superclass* of the object, then we can still use this compareTo method to compare elements in the array. For example, if we have:

```
public class Dog implements Comparable<Animal> { /* ... */ }
```

The `Dog` class has defined what it means for itself to be compared to other animals. However, the `Dog` is an `Animal`, so we can compare two dogs against each other (although we are unable to use the `Dog`-specific parts of the other `Dog` object in the comparison). We can reflect this desire—we want types that can be compared with each other or themselves and a superclass—with a *generic wildcard*.

```
public static <T extends Comparable<? super T>> void insertionSort(T[] arr) {
    for (int i = 0; i < arr.length; i++) {
        for (int j = i; j < arr.length; j++) {
            if (arr[j].compareTo(arr[i]) < 0) {
                T temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }
    }
}
```

The `? super T` clause introduces a *wildcard* (denoted by the `?`) and puts a *lower bound* on that wildcard. The effect is that we are further restricted in the type that is used used to instantiate the Comparable interface. The method takes an array whose carrier type `T` implements the Comparable interface against types that are `T` or a superclass of `T`.

Like subtyping of generic type parameters, wildcards and bounds get very tricky. Here are your options:

- `?` specifies an unconstrained wildcard generic type. Such a generic type can be instantiated with any type.
- `? super T` specifies a wildcard generic type with an *lower bound* of `T`. Such a generic type must be instantiated with `T` or a superclass of `T`.
- `? extends T` specifies a wildcard generic type with a *upper bound* of `T`. Such a generic type must be instantiated with `T` or one of its subclasses.

To deterimine which of these wildcards to use in the arguments of methods that we write, we must consider how the argument to the method is used:

- Does the method use the parameter as an data source? We call such a parameter an *in parameter*.
- Does the method use the parameter as a data sink or output? We call such a parameter an *out parameter*.

For our sorting algorithms above, while the input array is an "input", we are really changing the structure of the array (by sorting it). Therefore, the array functions as an out parameter. Once we have identified for each parameter whether it is an in or out parameter, then for type parameters found in the parameter's corresponding type, we choose the following:

- For *in parameters*, we use an upper bound wildcard: `? extends T`.
- If the *in parameter* does not have any particular type constraints, *i.e.*, we only use it as if it was an `Object`, then use an unbounded wildcard: `?`.
- For *out parameters*, we use a lower bound wildcard: `? super T`.
- For parameters that we consider both *in* and *out*, we do not use a wildcard. Instead we use a concrete type instead.

As an example, consider a method that performs a generic copy from a source List to a destination List.

```
public static <T> void copy(List<? extends T> src, List<? super T> dst) { /* ... */ }
```

The `src` parameter is the source of the copy, so it is an *in parameter*. In contrast, the `dst` parameter is the destination of the copy, so it is an *out parameter*.