Sorting 8

In this chapter, we will study sorting—how to arrange a given sequence in a certain order, e.g., from small to large. For a concrete example, consider the following array of numbers:

```
s = [54, 26, 93, 17, 77, 31, 44, 55, 20]
```

We want an efficient means to create a sorted version of this array, producing

```
t = [17, 20, 26, 31, 44, 54, 55, 77, 93]
```

The numbers in the resulting array are arranged from small to large, and we term this process sorting. In general, we want to sort generic objects, not just integers. The most common ordering is based on numerical order (i.e., ordering them according to some numerical value), lexicographical order (i.e., ordering them like in a dictionary), or some combination of them—although the users can specify custom ordering to suit their needs.

Sorting. How should we rearrange a given array of numbers so they appear from small to large in the fastest time possible? This question is fundamental to computing and has far-reaching applications. For example, after sorting, similar items will be close together in the array, allowing for easy duplicate removal or finding similar items.

We will start by modeling in Java how two objects can be compared. Based on this, we set out to test whether a given sequence is already sorted from small to large. This computation turns out to inspire a natural sorting algorithm, bubble sort. Following that, we will explore other sorting algorithms. The simpler sorting algorithms appear to all have $O(n^2)$ running time, but it is possible to break this quadratic running time bound. In the final sections of this chapter, we will look at faster algorithms that run in $O(n \log n)$ time—merge sort and quicksort—both divide-and-conquer algorithms

8.1 Rules of The Game

Sorting algorithms described in this chapter are known as comparison-based sorting algorithms because they rely on the premise that we can compare two objects `x` and `y` and know whether `x < y`, `x == y`, or `x > y`. For example, we can compare any two integers and determine their relative ordering in this fashion. So could we compare any two `String`s.

But we cannot always meaningfully compare any two objects. To require that the objects can be compared, we work with an interface called `Comparable`, which offers a comparison function. In particular, the sorting algorithms in this chapter assume a data type `T` such that `T extends Comparable<T>`. Such a data type has the following property: `x.compareTo(y)` returns an `int` with one of the following outcomes:

For example, if `T` is such that `T extends Comparable<T>`, we can implement a function `boolean less(T x, T y)` that tests whether `x < y` as follows: `return x.compareTo(y) < 0;`

- a positive value (> 0) means `x > y`.
- zero ($= 0$) means `x == y`.
- a negative value (< 0) means `x < y`.

For sorting to be well-defined, we implicitly assume that the objects and the comparison function form a *total order*. This is a mathematical property asserting that for all x, y, z, (i) if $x \leqslant y$ and $y \leqslant x$, then $x = y$; (ii) if $x \leqslant y$ and $y \leqslant z$, then $x \leqslant z$; and (iii) it holds that $x \leqslant y$ or $y \leqslant x$. Readers are invited to check that these properties hold for integers and strings.

To avoid having to declare the type bounds too often, we will implement functions (i.e., methods) in this chapter inside the following class declaration:

```
public class Sorting<T extends Comparable<T>>
```

which makes `T` inside the class to be `T extends Comparable<T>`. Hence, inside this class, any object of such type `T` is guaranteed to have a `.compareTo` as discussed earlier.

To exchange `a[i]` and `a[j]` in a Java array:

```
T temp = a[i];
a[i] = a[j];
a[j] = temp;
```

We also define a utility function `void swap(T[] a, int i, int j)` inside this class. This function exchanges the contents of `a[i] a[j]`, as shown next to this paragraph.

Testing for Sortedness

As a quick example to demonstrate the setup, as well as to motivate how sorting algorithms are designed, we ask, how can one write a program to check whether a given array of objects is already sorted from small to large? We will write a function

```
boolean isSorted(T[] a)
```

which takes as input an array of type `T` elements and returns a Boolean indicating whether `a` is already sorted from small to large.

The idea to solve this problem is a simple one: check if all adjacent elements are in the right order (i.e., `a[i] <= a[i+1]`). If the array is already sorted, every adjacent pair will be in the right order. To make such comparisons, we will resort to `.compareTo` as follows:

```
boolean isSorted(T[] a) {
    int n = a.length;
    for (int i=0;i<n-1;i++) {
        if (a[i].compareTo(a[i+1]) > 0)
            return false; // found an offending pair
    }
    return true; // sorted! no offending pairs
}
```

Note that in this code, if the pair `a[i]` and `a[i+1]` is in the right order, we expect `a[i].compareTo(a[i+1]) <= 0`. A violation is, therefore, the opposite—hence, we check if `a[i].compareTo(a[i+1]) > 0`.

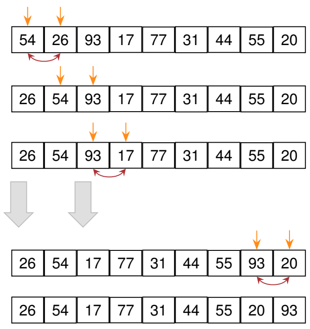
Example. It helps to step through the execution of `isSorted` on concrete inputs. If we run `isSorted` on the array `s` from the beginning of the chapter, the code will find an "offending pair" at `i=0` and it quits after the first pair. On the other hand, if we run it on the array t also from the beginning of the chapter, the for-loop will progress till completion without finding any out-of-whack pair and the code returns `true`.

8.2 Bubble Sort

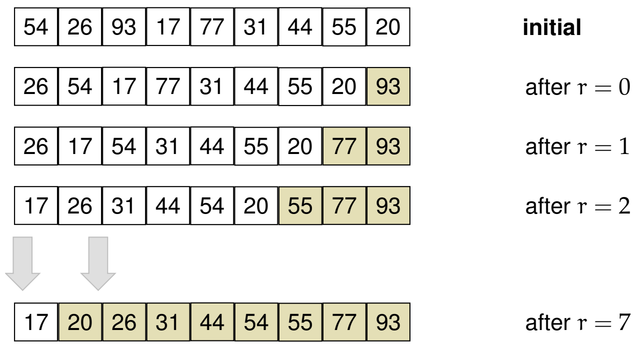
The first sorting algorithm we will look at is a simple one. To motivate it, think back to our algorithm for testing if a given array is already sorted. If the array is not sorted, the code will find an adjacent pair where `a[i] > a[i+1]`. We can fix this by swapping them. Although it is unclear a priori if this will help anything overall, it is intuitively clear that it fixes one violation.

Building on this idea, the bubble sort algorithm makes passes over the input sequence. In each pass, it compares adjacent elements and swaps them if they are out of order. We know the sequence is sorted when we are not making any more swaps. However, it is less clear how many passes will be needed or whether this idea will lead to a sorted sequence eventually.

Example: One Pass. Below shows the first pass through the sequence s from the beginning of the chapter:



In our example, after one pass, the largest element (**93**) is already in the right location. This is not coincidental. In general, an important observation is that for each pass through the sequence, the next largest value will be in the proper spot. In fact, we can say that each element floats, or "bubbles", up to the correct location. This leads to the invariant that after $r = 0, 1, 2, \ldots$ passes, the $(r+1)$ rightmost spots store the $(r+1)$ largest elements of the whole sequence. We illustrate this observation below, where the sequence after each pass is shown and the $(r+1)$ largest elements are shaded:



How many passes through the sequence are necessary then? It is easy to see that at most $n-1$ passes are needed. More specifically, after $n-1$ passes, our invariant guarantees that the $(n-1)$ largest elements are in the right places. But then, this means that one last element has nowhere else to be but to be in the right spot as well.

Turning this into code is straightforward, as shown in Code 8.1. Furthermore, since pass r takes $O(n-r)$ time and the code performs at most $n-1$ passes, the running time is $O(n^2)$. As an optimization, we could stop the code as soon as the algorithm makes no swaps during a pass.

$$(n-1)+(n-2)+\cdots+1 \\ = O(n^2)$$

Code 8.1: Bubble Sort.

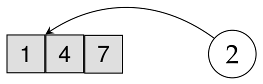
```java
void bubbleSort(T[] a) {
    int n = a.length;
    for (int r=0;r<n-1;r++) {
        for (int i=0;i<n-r-1;i++) {
            if (a[i+1].compareTo(a[i]) < 0)
                swap(a, i, i+1);
        }
    }
}
```

8.3 Insertion Sort

We will now look at another implementation that is equally intuitive. Insertion sort builds the sorted sequence one at a time. It rests on the following idea—inserting a new item into an already-sorted sequence to keep it sorted involves just two easy steps:

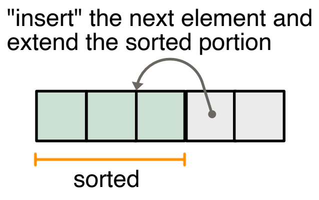
1. Locate where the new element should be; and
2. Make room for the new element by moving/rearranging the existing sequence.

As an example, consider the following example, in which 2 is being inserted into an already-sorted sequence.



To know that the number 2 has to go between 1 and 4, the algorithm can simply scan the sequence from the right end until a snug-fit spot is found. More generally, if x is the new element, we are looking for a position where the value of x is between the value to the left and the value to the right. It is possible that x has nothing to the left because it is the smallest element so far or that x has nothing to the right because it is the largest element so far.

Insertion Sort. At a glance, insertion sort performs the following steps in each iteration:



In terms of running time, this process alone could take linear time in the worst-case, having to look through the whole sequence. More sophisticated algorithms (e.g., binary search) can reduce the running time. But this is often not useful because in the common representation of a standard array, the next step of making room for the new element requires linear time nonetheless.

A standard implementation works directly on the input array or a copy of it. This is convenient because for each iteration $i = 1, 2, \ldots, n-1$, the invariant at the end of that iteration is that

> The elements `a[0], a[1], ..., a[i]` are the correct sorted arrangement of the first `i+1` elements.

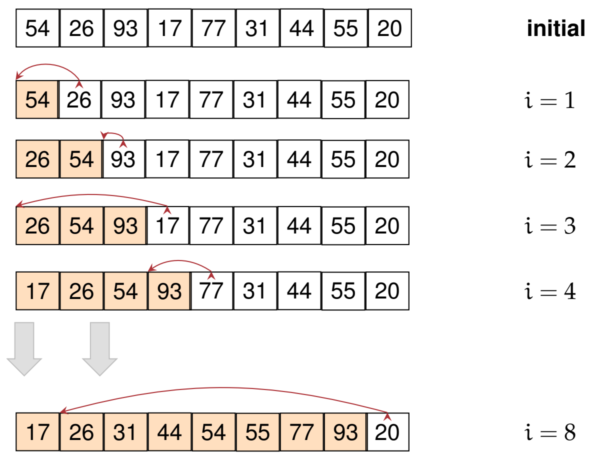
Hence, the role of iteration `i` is to insert the value of `a[i]` into an already-sorted sequence `a[0], a[1], ..., a[i-1]`.

We illustrate the working of the insertion sort algorithm below. The elements shaded in gray are those the invariants guarantee to be sorted already.

Code 8.2: Insertion Sort.

```java
void insertionSort(T a[]) {
    int n = a.length;
    for (int i=1;i<n;i++) {
        // invariant: a[0]...a[i-1] is sorted
        T elt = a[i];
        int j=i;
        while (j>0 && elt.compareTo(a[j-1]) < 0) {
            a[j] = a[j-1];
            j--;
        }
        a[j] = elt;
        // invariant: a[0]...a[i] is sorted
    }
}
```

Notice that the iteration number i is also the demarcating point between the sorted portion of the array and the unprocessed elements.



It may appear that we will need two separate passes—one to locate where the new element will be and the other to rearrange the elements. Rather neatly, they can be combined into one. If `elt` is the new element being inserted, the fact that an element `a[j] > elt` means that `a[j]` should be moved one position to the right. In this way, we can simultaneously find the right spot for `elt` and shift the elements to make room for it. Turning this into code is straightforward, as shown in Code 8.2. Because each iteration i takes $O(i)$ time in the worst case, the worst-case running time of insertion sort is $O\left(\sum_{k=1}^{n-1} k\right) = O(n^2)$. This makes it another algorithm, alongside bubble sort, that requires quadratic time in the worst case.

Code 8.3: Selection Sort.

```
void selectionSort(T a[]) {
    int n = a.length;
    for (int i=0;i<n;i++) {
        // invariant: a[0]...a[i-1] are the i smallest elements
        int minDex=i;
        for (int j=i+1;j<n;j++)
            if (a[j].compareTo(a[minDex]) < 0)
                minDex=j;
        if (minDex!=i)
            swap(a,minDex,i);
        // invariant: a[0]...a[i] are the i+1 smallest elements
    }
}
```

8.4 Selection Sort

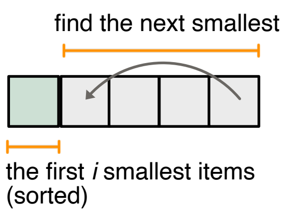
This time we will briefly look at another sorting algorithm that, like insertion sort, builds the sorted sequence one at a time. In a typical implementation, like insertion sort, it maintains a sorted portion of the sequence and an unprocessed portion. However, unlike insertion sort, it uses a different idea to grow the sorted portion—selecting the smallest item from the unprocessed portion to add to the sorted portion. In this sense, selection sort can be seen as carrying out the following steps:

- first, find the minimum element; call this e_0
- then, find the minimum element after excluding e_0; call this e_1
- then, find the minimum element after excluding e_0 and e_1; call this e_2;
- and so on.

A standard implementation works directly on the input array or a copy of it. For each iteration $i = 0, 1, 2, \ldots, n-1$, the invariant at the end of that iteration is that

> The elements `a[0], a[1], ..., a[i]` are the $i+1$ smallest elements.

Selection Sort. At a glance, selection sort performs the following steps in each iteration:



Therefore, the goal of the i-th iteration is to find the smallest element from among `a[i], a[i+1], ..., a[n-1]`. This is easy to implement and requires considering $n-i$ elements, hence taking $O(n-i)$ time.

Code 8.3 shows an implementation of insertion sort. The inner loop finds the index of the next smallest element, as described earlier. The index `maxDex` is then swapped with `i`, satisfying the invariant. The overall running time is $O(\sum_{k=0}^{n-1}(n-k)) = O(\sum_{k=1}^{n} k) = O(n^2)$.

8.5 Breaking The Quadratic Barrier

While the sorting algorithms considered so far are intuitive and easy to implement, they all require quadratic time—aka. $O(n^2)$ time—in the worst case. It is natural to wonder if faster algorithms are possible.

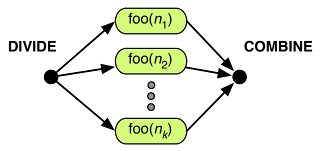
To break the quadratic barrier, we now discuss a new technique. Divide and conquer is a highly versatile technique that generally lends itself well to the design of fast algorithms. We have already seen the divide-and-conquer technique many times without knowing its name. This time we'll give the technique a label and look at a few examples.

The structure of a divide-and-conquer algorithm follows the structure of a proof by (strong) induction. This makes it easy to show correctness and also to figure out the running time. The general structure looks as follows:

— **Base Case:** When the problem is sufficiently small, we return the trivial answer directly or resort to a different, usually simpler, algorithm, which works great on small instances.

— **Inductive Step:** First, the algorithm **divides** the current instance I into parts, commonly referred to as *subproblems*, each smaller than the original problem. Then, it **recurses** on each of the parts to obtain answers for the parts. In proof, this is where we assume inductively that the answers for these parts are correct. Finally, based on this assumption, it **combines** the answers to produce an answer for the original instance I.

Divide and Conquer. To solve a problem `foo` on a instance of size n, a divide-and-conquer process (schematically depicted below) divides the instance into smaller instance of sizes $n_1, n_2, \ldots, n_k$, where k is typically at least 2, recursively solves each of these pieces, and combines their results into the solution to the original problem of size n:



So far, this process is identical to what we have seen as our recipe for designing recursive algorithms. The signature of divide-and-conquer algorithms is that we attempt to make each part a sizable fraction of the current problem—not just one smaller or a constant smaller.

8.6 Merge Sort

Almost invariably, the main idea in reducing the time complexity so far has been to aggressively reduce the problem size: whereas both insertion sort and bubble sort, in essence, reduce the problem size by 1, we wish to do better, perhaps cutting the size down in half—like we did for powering b^w. Following that pattern, we will attempt to write a recursive algorithm that splits a given sequence in half.

The earliest variant of merge sort dated back to 1945 and is often attributed to John von Neumann. To reconstruct merge sort using the divide-and-conquer strategy, two questions need to be answered:

(Q1) **How to solve small instances?** As with our previous problems, we measure the size of the input by the length of the input sequence.

Do we know how to sort an empty sequence and a sequence of length 1? This should be easy. Both the empty sequence and a sequence of length 1 are already sorted. Hence, we can simply return the input sequence.

(Q2) **How to tackle an instance in terms of smaller instances?** Say, inside the call where the input has length n, we already know how to sort any input sequence of length less than n. How can we use this to our advantage?

Let's break the input sequence into two (roughly) equal-sized sequences—`left` and `right`—at midpoint.

Furthermore, both these sequences are shorter than n, in fact, just half of that. Hence, we could call ourselves recursively on both of them, yielding a sorted copy of `left` and `right`, respectively.

What we would like to do is to combine these two sorted sequences into a single one that is sorted. We hope to be able to do this efficiently. The logic for merging two such sorted sequences will be captured by a `mergeInto` function.

Pictorially, we have a situation such as in Figure 8.1. Notice that our job is not yet over—we have two sorted sequences and we need to merge them into a single fully-sorted sequence.

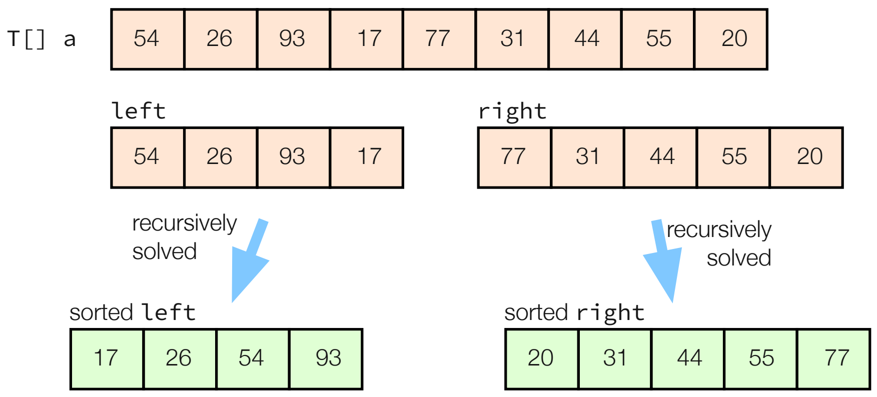


Figure 8.1: Merge sort—divide and recurse, but how to merge?

In code, our discussion so far can be summed up as follows. To avoid a dark corner of Java concerning generic arrays, we opt to use a destination-passing style, where we offer each function a space to write into rather than relying on it to allocate the space where the result is returned.

Code 8.4: Merge Sort (Main Function)

```java
void mergeSort(T[] a) {
    int n = a.length;
    if (n <= 1) return ; // a is already sorted

    T[] left  = Arrays.copyOfRange(a, 0, n/2);
    T[] right = Arrays.copyOfRange(a, n/2, n);

    mergeSort(left);
    mergeSort(right);
    mergeInto(left, right, a);
}
```

From this code, each call on an array of length n makes *two* calls recursively on arrays of length $n/2$. Let $T_{\texttt{mergeInto}}(x, y)$ denote the running time of `mergeInto` on array sizes x and y, and it is evident that the running time recurrence for merge sort is

$$T(n) = 2T(n/2) + O(n) + T_{\texttt{mergeInto}}(n/2, n/2),$$

where $T(0) = T(1) = O(1)$, and the $O(n)$ term comes from allocating and copying the `left` and `right` arrays. The last piece of our puzzle is therefore the merging of two sorted arrays.

Merging Sorted Arrays

Because the two arrays that we are merging are already sorted, we can "peel off" the front of either array, whichever is smaller. To implement this, we keep two fingers, one on each array, pointing to the first index we have not peeled off. We compare the elements where the fingers are pointing and peel off the smaller one. Eventually, one array will be empty. When we reach this point, we will simply copy the elements from the remaining nonempty array into the output.

Code 8.5 (below) is an implementation of this idea. The `mergeInto` will read from sorted arrays `a[]` and `b[]`, and write the merged sequence to `out`. The fingers into `a` and `b` are `i` and `j`, respectively.

Code 8.5: Merging Two Sorted Arrays

```
void mergeInto(T[] a, T[] b, T[] out) {
    int i = 0, j = 0;
    for (int o=0;o<out.length;o++) {
        if (i >= a.length)
            out[o] = b[j++];
        else if (j >= b.length)
            out[o] = a[i++];
        else if (a[i].compareTo(b[j]) < 0)
            out[o] = a[i++];
        else
            out[o] = b[j++];
    }
}
```

Example. Consider merging `a=[17, 26, 54, 93]` and `b=[20, 31, 44, 55, 77]`.

Initially
```
a=[17, 26, 54, 93]
b=[20, 31, 44, 55, 77]
out=[]
```

Iteration #1:
```
a=[26, 54, 93]
b=[20, 31, 44, 55, 77]
out=[17]
```

Iteration #2:
```
a=[26, 54, 93]
b=[31, 44, 55, 77]
out=[17, 20]
```

Iteration #3:
```
a=[54, 93]
b=[31, 44, 55, 77]
out=[17, 20, 26]
```

Iteration #4:
```
a=[54, 93]
b=[44, 55, 77]
out=[17, 20, 26, 31]
```

Iteration #5:
```
a=[54, 93]
b=[55, 77]
out=[17, 20, 26, 31, 44]
```

$\vdots$

Final:
```
a=[]
b=[]
out=[17,20,26,31,44,54,55,77,93]
```

Since the length of the output array is the sum of the lengths of the input arrays, this code runs in time $O(n + m)$, where n is the length of `a` and m is the length of `b`. Notice that each iteration of the for-loop takes constant time.

Running Time Analysis

We have previously established that merge sort takes constant time on arrays of length at most 1, and for $n > 1$, the recurrence is

$$T(n) = 2T(n/2) + O(n) + T_{\texttt{mergeInto}}(n/2, n/2).$$

Together with our analysis of `mergeInto`, the recurrence becomes

$$T(n) = 2T(n/2) + O(n),$$

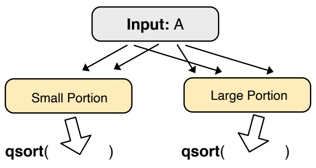
since `mergeInto` takes time $O(n/2 + n/2) = O(n)$. This is a standard recurrence, which solves to $O(n \log n)$. In conclusion, we have broken the quadratic barrier: merge sort runs in $O(n \log n)$, a marked improvement over the algorithms discussed earlier in the chapter.

8.7 Quicksort

Quicksort, invented by Tony Hoare in 1956, is another divide-and-conquer algorithm. Compared to merge sort, which has a simple divide step at the expense of a nontrivial combine step, quicksort is an algorithm with a trivial combine step perhaps at the cost of a more complex divide step.

We will aim for an extremely simple combine strategy: If the recursive solutions are X and Y, we want to simply concatenate them.

```
def qsort(A):
    if len(A) <= 1:
        return A
    # do something to derive X, Y
    return X + Y
```



What strategy could we use to divide up the problem that will satisfy this requirement? Because we wish to simply concatenate the parts, this dictates what the parts have to look like relatively to each other. Importantly, each element in X must be less than every element in Y.

Split At a Pivot Value. One way to satisfy the requirements above is the following simple idea: Pick a value called the pivot p, and split the input sequence A into *three* parts:

- `lt` — elements that are strictly less than p;
- `eq` — elements that are exactly p; and
- `gt` — elements that are strictly greater than p.

The `eq` part is there for reasons that will soon be apparent. We can quickly implement this using Java’s support for variable-sized arrays (e.g., `ArrayList`). Again, we will continue to use the destination-passing style.

Code 8.6: Splitting at a pivot value.

```
void splitInto(List<T> a, T p,
           List<T> lt, List<T> eq, List<T> gt) {
    for (T elt : a) {
        int cmp = elt.compareTo(p);
        if (cmp < 0)        { lt.add(elt); }
        else if (cmp == 0)  { eq.add(elt); }
        else                { gt.add(elt); }
    }
}
```

It is clear that `splitInto` runs in $O(n)$ time, where n is the length of the sequence `a`.

Good Pivot? Observant readers may already notice that as long as the pivot comes from an element of the input array, each of `lt` and `gt` is guaranteed to be shorter than the input itself. However, as it turns out, the pivot choice crucially determines the performance of quick sort. For starters, we will use a naïve strategy of using the front element as the pivot, resulting in the following code:

Code 8.7: Quicksort Using the First Element as Pivots.

```java
void qsort(List<T> a) {
    if (a.size() <= 1) return ;

    List<T> lt = new ArrayList<>(),
        eq = new ArrayList<>(),
        gt = new ArrayList<>();

    T p = a.get(0);
    splitInto(a, p, lt, eq, gt);

    qsort(lt); qsort(gt);

    // clear a & concatenate them all
    a.clear();
    a.addAll(lt); a.addAll(eq); a.addAll(gt);
}
```

This attempt at writing quicksort was a success in that the algorithm correctly sorts a given input sequence; however, from a performance point of view, it is not great. To see this, we will write a recurrence for the algorithm:

$$T(0) = T(1) = 1$$
$$T(n) = T(n_{lt}) + T(n_{gt}) + O(n),$$

where $n_{lt} = \texttt{len}(\texttt{lt})$ and $n_{gt} = \texttt{len}(\texttt{gt})$.

If n_{lt} and n_{gt} were about $n/2$, the recurrence would solve to $O(n \log n)$, as desired. However, this is not always the case. It is not too hard to convince ourselves that if the input is already sorted, e.g., `a = {1, 2, 3, 4, 5}`. This implementation will suffer because each time it divides the list into `lt={}`, `eq = {1}`, and `gt` being the rest, resulting in $T(n) = T(n-1) + O(n)$, which is an $O(n^2)$ algorithm.

How can we fix this problem? Various strategies for picking pivots can be tried. Use the last element? Use the middle element? Use the median of first, middle, and last? Indeed, they have been extensively studied. But as it stands, no matter which fixed position we deterministically choose the pivot to be, an adversary can force our hands so this algorithm runs slowly.

An effective—yet extremely simple—strategy turns out to be to pick the pivot uniformly at random from the input array. In a way, by making random choices, the adversary cannot know ahead of time which pivots we are using, so it cannot construct an example that is truly bad for our code.

Randomized Quicksort

We will write code to pick the pivot randomly from the input sequence. But how can we make random choices? For this, we will need help from a random-number generator (RNG), which most languages provide.

Java offers a random-number generator class `java.util.Random`. An instance of the class only needs to be created once and can be used multiple times. The best practice is therefore to declare such an instance as a member variable, like so:

```
Random RNG = new Random(); // declared as a member variable
```

Inside the quicksort code, the only line that we will change is the line that picks the pivot. Instead of using the pivot from a fixed index, we will replace that line with the following code:

```
T p = a.get(RNG.nextInt(a.size()));
```

The `nextInt(n)` method returns a random integer between 0 and $n-1$, hence allowing this code pick the element at a random index.

This means our quicksort implementation with randomly-chosen pivots works as depicted below:

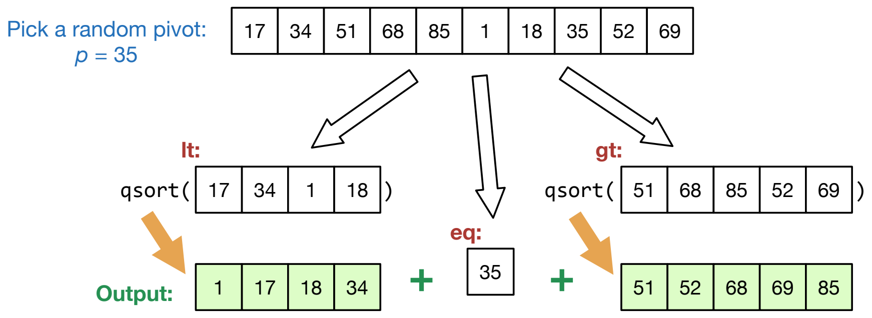


Figure 8.2: Schematic depiction of randomized quicksort execution

How fast is this version of quicksort? We cannot fully answer this question at this point. In Chapter 11, we will study the techniques for analyzing this type of code and revisit this algorithm in Section 11.5. For intuition, notice that in reality, the split is unlikely to be perfect 50:50; however, it cannot be so uneven, either. Overall, the behavior is not too far off from $T(n) = 2T(n/2) + O(n)$, so randomized quicksort runs in $O(n \log n)$ time under some notion of probabilistic time.

8.8 Java Built-in Sorting

Java, as with most modern languages, has built-in capabilities for sorting an array/collection. We will discuss the following two utility functions:

- `Collections.sort` for sorting a collection such as an `ArrayList`.
- `Arrays.sort` for sorting a primitive array.

Readers are recommended to visit the Java documentation for the detail of these functions. For now, a few quick examples are in order.

Example. If we have an array `int[] a`, we can sort it like so:

```
int[] a = {3, 5, 1, 9, 8};
Arrays.sort(a);
```

If we have an `ArrayList<Integer>` (which is a collection), we can sort it like this:

```
ArrayList<Integer> a = new ArrayList<>(List.of(3, 5, 1, 9, 8));
Collections.sort(a);
```

Sorting By a Key Function. In many situations, we wish to sort a sequence of elements according to some order other than the "natural" order. For example, we have an array

```
String[] colors = { "violet", "red", "blue", "teal",
  "green", "pink", "magenta", "orange" }
```

If we just sort it using `Arrays.sort`, we would get the array

```
{"blue", "green", "magenta", "orange", "pink",
  "red", "teal", "violet"}
```

where the elements are lexicographically ordered.

Suppose, instead, we want to sort the colors by the length of the color name. How can we do that? As it turns out, we could specify how we wish the data elements to be ordered by, by giving the sort function an extra parameter—a function for comparing a pair of elements.

```
Arrays.sort(colors,
    (String x, String y) -> x.length() - y.length());
```

This extra parameter is called the `Comparator`. In short, the comparator of `x` and `y` describes how we wish `x` and `y` to be compared. The function should return -1, 0, and $+1$ in the same way a `compareTo` would.

8.9 Applications: Sort/Merge-Inspired Algorithms

Many problems do not look anything like a sorting problem at first glance. Nonetheless, the sheer fact that the instance is sorted and in some cases, a merge-like routine can be of great help in solving them. Here, we will look at two examples.

Duplicate Removal

In our first example, we are given a sequence of elements (think about integers if it helps you visualize better) and we want to produce a new sequence where each unique element in the original sequence shows up exactly once—that is, removing all the extra copies. The output can be in any order.

Example. If the input is `[3,7,3,8,8,7,1,4,3]`, one possible output is `[3,7,8,1,4]`. To see what we mean by duplicate removal, notice how `3` appears only once in the output even though it appears three times in the input.

Code 8.8: Duplicate removal on a comparable sequence.

```java
import java.util.*;

<T extends Comparable<T>> List<T> removeDuplicate(List<T> xs) {
    List<T> elements = new ArrayList<>(xs);
    Collections.sort(elements);
    List<T> out = new ArrayList<>();

    T prev=null;
    for (T elt : elements) {
        if (!elt.equals(prev)) {
            prev = elt;
            out.add(elt);
        }
    }
    return out;
}
```

The readers might wish to think about this for a moment. (*Hint:* Things look nicer when sorted.)

One way to solve this problem is to realize the following:

> *The Crux:* Once sorted, all copies of the same element are next to each other.

To illustrate this observation, notice that on the example input, sorting `[3, 7, 3, 8, 8, 7, 1, 4, 3]` yields `[1, 3, 3, 3, 4, 7, 7, 8, 8]`, which makes it easy to tease out unique elements.

Turning this idea into code is not difficult, as shown in Code 8.8. In the code, we rely on `Collections.sort` to sort the sequence, but to make this all work, the elements in the sequence have to comparable, as enforced by the type bound `T extends Comparable<T>`. Chapter 12 discusses another effective means to solve the same problem.

Intersection

We will now turn our attention to another problem. Let A and B be two sequences. Assume for now that they individually contain no duplicates, an assumption we can conveniently satisfy using our previous example. We are interested in producing a sequence which is the intersection of the two sequences—that is, an element will be there if it is in both A and B. The result can be in any order.

Example. As a few examples:

- On input `A = [1, 4, 2]` and `B = [7,1,9,5,2]`, one possible output is `[1, 2]`.
- On input `A = [1, 5, 3]` and `B = [6, 4, 2]`, the output is `[]`.

Code 8.9: Count the number of common elements.

```java
// Assume: a and b must be sorted ascendingly
int countIntersectSorted(int[] a, int[] b) {
    int common=0, i=0, j=0;

    while (i < a.length && j < b.length) {
        if      (a[i] == b[j]) { common++; i++; j++; }
        else if (a[i]  < b[j]) { i++; }
        else                   { j++; } // i.e., a[i] > b[j]
    }

    return common;
}
```

In this example, we will in fact solve a slightly simpler problem: find the number of elements in the intersection (i.e., count the number of common elements). How to solve this problem quickly? At first glance, it has nothing to do with sorting or merging. But one thing we notice when we wrote the merge routine was the following:

Identical elements from different sides are compared

So, we are going to take advantage of this observation, as shown in Code 8.9.

Exercises

Exercise 8.1. Produce a visual trace of the execution of insertion sort on the following input:

```
4, 2, 1, 3, 9, 8, 6, 7, 5
```

Exercise 8.2. Produce a visual trace of the execution of selection sort on the following input:

```
4, 2, 1, 3, 9, 8, 6, 7, 5
```

Exercise 8.3. If the input sequence is already ordered from small to large (i.e., sorted), how many comparisons is insertion sort going to make? How about bubble sort and selection sort?

Exercise 8.4. Using the fewest number of comparisons, write a program that sorts 5 distinct integers. For concreteness, write a function

```java
int[] sort5(int a, int b, int c, int d, int e)
```

that returns a length-5 array that stores `a, b, c, d, e` in sorted order.

Exercise 8.5. Give the best- and worst- case running time of each of the algorithms discussed in this chapter.

Exercise 8.6. Consider a quicksort implementation that always chooses the middle element (i.e., index $\lfloor n/2 \rfloor$). For $n \geqslant 1$, design a sequence of integers that if given as input will result in quicksort taking $\Theta(n^2)$.

Exercise 8.7. In a remote village known as Salaya, zombies and humans have lived happily together for many decades. In fact, no one can quite tell zombies and humans apart. However, when these "people" line up in a single row, all sorts of trouble ensue, including this weird phenomenon: human beings will line themselves up from tall to short, but zombies act erratically.

In particular, if `line` is an array of heights of the population of this village, we would expect that `line`$[i] \geqslant$ `line`$[j]$ for $i \leqslant j$. But this simply isn't true in many cases especially with zombies around. Hence, one nobleman—or is he a zombie?—came to you for help: he wants to know how many pairs of his people violate this social norm.

Your Task: Write a function `int countBad(int[] hs)` that takes an array of n numbers and returns the number of pairs $0 \leqslant i < j < n$ such that `hs`$[i] <$ `hs`$[j]$ (i.e., the number of pairs that violate the social norm).

For example (abusing Java's array notation):

- `countBad({35, 22, 10}) == 0`
- `countBad({3,1,4,2}) == 3`
- `countBad({5,4,11,7}) == 4`
- `countBad({1, 7, 22, 13, 25, 4, 10, 34, 16, 28, 19, 31}) == 49`

Performance Expectations: We expect your code to run in at most $O(n \log n)$ time, where n is the length of the input array.

(Hint: Write a `merge`*-like algorithm that computes two things: (1) the combined sorted sequence and (2) the number of out-of-wack pairs. Your complete solution should look almost identical to the merge sort algorithm except it computes this additional thing.)*

Exercise 8.8. Implement merge sort that takes a `Comparator<T>` as an input parameter instead of relying on `.compareTo`.

Exercise 8.9. Implement quicksort that takes a `Comparator<T>` as an input parameter instead of relying on `.compareTo`.

Exercise 8.10. Let `a` and `b` be two sorted arrays of integers. Each array only stores unique numbers. Write a function

```
int countUnion(int[] a, int[] b)
```

that computes the total number of unique integers across the two arrays (i.e., the size of their union).

Chapter Notes

As a fundamental problem in computing, sorting has undoubtedly been extensively studied. It was tackled as early as in 1951 around the time the first general-purpose digital computer was starting to be programmed. Simple exchange-based algorithms such as bubble sort, insertion, and selection sort are pretty much folklore. An information-theoretic argument can be used to show a lower bound of $\Omega(n \log n)$, meaning no comparison-based sorting algorithms can use fewer than $\sim n \log n$ comparisons in general [Cor+09; KT06]. This means that algorithms of this type necessarily need at least $n \log n$ time in general. Such time-optimal algorithms were, in fact, known from early on. Both merge sort and quicksort can achieve $O(n \log n)$ running time. According Donald Knuth [Knu98], merge sort was invented by John von Neumann around 1945. Quicksort [Hoa61] was invented by Tony Hoare in 1956, though unpublished until 1961. In general, the memory usage of a sorting algorithm tends to be linear in the input size.

This chapter is concerned with basic implementations and analysis of standard sorting algorithms. Various practical techniques for implementing them are described in the literature but are beyond the scope of this chapter. Other popular sorting algorithms include heap sort, shell sort, and tim sort. While comparison-based sorting algorithms cannot be faster than $n \log n$ in general, other sorting strategies can sometimes get around this lower bound, for instance, by exploiting some special structure of the data. For example, it is possible to sort in $O(n)$ time an array of `0`s and `1`s of length n. Radix sort and counting sort are examples of algorithms in this category. More advanced textbooks on algorithms (e.g., [Cor+09; KT06; SW11]) describe what we study here and more in further depth.