Chapter 4: Sorting

Introduction

Technically, sorting is a "search problem" where we test every candidate solution to determine if it meets our requirements and stop when we find a valid solution (by applying a "decision problem" to each). A naïve solution would entail, for each of the $N!$ possible permutations, comparing every pair of neighboring elements to see if they are "in order." This "brute force" method is essentially the technique behind the infamous Bogosort. Its runtime grows exponentially with the size of the input (don't try this at home).

Let's go even further back to consider what it means to say that two elements are in order. What's an element? It's some kind of *thing*—we'll consider it to be an object—that supports the notion of ordering. If these objects represent numbers, then we're good to go, provided that those numbers are scalars (each having only one value). Complex numbers, vectors, matrices, etc. aren't readily orderable. There are no debates about the order of the (scalar) "counting" numbers because they're used for, well, counting. If you have four apples and I have five apples, then I win—five is greater than four. Rational numbers can also be easily compared in this way. Irrational numbers? Transcendental numbers? Maybe, but let's not get too sidetracked. If you're really interested in this concept of ordering, see Lexicographic Order.

Another type of object that is easily ordered is a word, in the sense of natural language. But which language? That's going to make quite a difference. In the (traditional, pre-1994) Spanish word order, for example, "*c"* and "*ch"* were considered different letters, lexicographically. Even today "*ñ"* is collated after "*n"*. If we have a collation scheme that defines the order of letters, we can order words in any language.

Given a domain of objects (e.g., the English words), If *any* two objects can be strictly ordered ($x < y$ or $x > y$), then we can say that the domain defines a (strict) Total Order. According to our earlier description of sorting, we should be able to sort objects from that domain. But what about objects that don't have a strict total order? If they have a non-strict total order, we're still OK for sorting but we will have to be able to deal with equality because there may be duplicates.

However, there's an entire class of objects which aren't orderable at all. How would we order colors, for example? Maybe we could order them according to the frequency of the light—that would give us a nice number that we could use. The snag is that light is generally a mix of frequencies. If we still want to sort colors, we're going to need to introduce the concept of a "key". A key is a bit like a "feature" in Machine Learning—it might be the object itself, or it might be some orderable attribute of the object—or even a function of several attributes. In the case of colors, we could define a key that suits our purpose, perhaps a number/name from a paint chart.

Let's define *comparable* as a property of an object such that we can order it against another object *by comparing their keys*. And we're going to require such a key to have at least a non-strict total order. So, when we compare the keys *x* and *y*, there will be one of three possible outcomes: $x < y$, $x = y$, or $x > y$. We will be able to navigate through our search space using the result. Note that, if we want a Boolean result (for a decision problem), we will need to combine these three results into two, for example, $x \leq y$, or $x > y$.

Counting Sorts

Whether we check the equality of two objects or, if orderable, we compare them, there will be the possibility of equality. That doesn't help us much in sorting. But people have been using equality as a way of ordering things for centuries. It's how filing systems work.

Since it's the oldest way of sorting (organizing) things, let's talk about it. Suppose you want to sort your friends by their eleven-digit international phone number. Let's start with the least-significant digit (0-9). We put all the 0s together, all the 1s, etc. on up to the 9s. But we're going to need a place to put them which has equal length as our input. What if the first one we need to copy (starting at the top or left of our list) is a 2. We'll need to know how many 0s and 1s there are and so put our 2 in the first empty slot thereafter. How do we know where that would be? We count them at the start. We'll need a place to put the 10 count values, too (actually, 11 spaces—one more). BTW, it should be clear that we're using the phone number as the key—but the objects are going to be in the form of key-value pairs (phone number, name). Obviously, when we copy from the input to the temporary storage (and back), we need to copy the whole object (k-v pair), not just the key.

If we're careful never to change the order of numbers with equal keys (we always copy them from the first un-copied number into the first available slot for that digit), then we have what's called a "stable" sort. A stable sort means that we don't disturb any existing order (see section "Stable Sorting" below). If we now do this for the other 10 digits of the phone numbers, we will eventually have a sorted list of phone numbers and friends.

How long will this take? If we have N friends, we must do N counting operations, and $2N$ copying operations. If there are D digits ($D = 11$ in our example), then the total amount of work performed is $\sim 3DN$. We must also determine the cumulative counts, but that takes time proportional to D (usually much smaller than N) so can be ignored for purposes of estimation (see "tilde" in Chapter 2).

$3D$ is a constant so the amount of work is linear with respect to the number of friends. Doesn't everyone say that sorting can't be done in linear time? Well, yes, but that applies to *comparison-based* sorting. Counting sorts like the one described above (called LSD radix sort) *are* linear in the number of objects. They are useful when an object is made up of a string of values, each of which is one of D possible values (D

could be as few as one but has no upper limit). For this reason, they're often referred to as *string sorts*.

We'll come back to string sorts later in the chapter 8 (Advanced Sorts).

Comparison Sorts

For short lists, counting (string) sorts are not the most efficient way of sorting and they're only useful when an object is a string. So, if we have a key available for a non-string object, we will use a comparison-sort.

As noted above, comparisons involve *pairs* of elements. How many pairs are in a collection of length N? It's the number of ways of choosing two objects from N objects, traditionally written $\binom{N}{2}$, one of the binomial coefficients. Its value is $\frac{N(N-1)}{2}$, which we can easily derive as follows: we have N possible ways of choosing the first element, but only $N-1$ ways remaining for choosing the second element. As far as invoking the comparison operator is concerned, it doesn't matter what order the elements are in (although the result cares). Therefore, we divide by two.

But that means there are approximately $\boldsymbol{O}(N^2)$ pairs! That would mean the algorithm runs in quadratic time! As we saw in chapter 2, quadratic algorithms do not scale well.

Still, if N is small, and/or we have a fast computer, we can stop worrying too much about the total time. Let's see where this idea leads.

Bubble Sort

Perhaps the most obvious sorting method is Bubble sort (in pseudo-code):

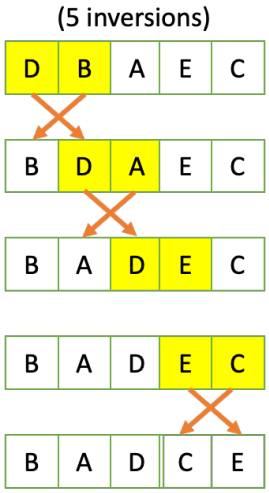
```
set w to point to the (non-existent) neighbor of the last element
while true {
    set x to refer to the "first" element of the collection*
    while x has neighbor y and y is not w
        conditionally swap x with y, and set x equal to y
    if the loop completed without swapping anything
        break
    set w to refer to the most recent value of x
}
```

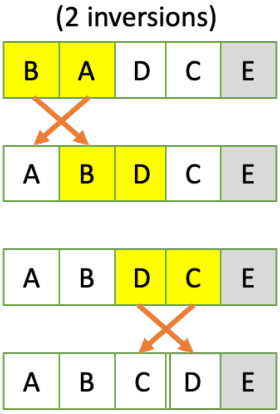
* We assume that the first element is at the left

Both loops have early exit possibilities. The inner loop exits when y reaches w. In other words, w represents the boundary between the elements on the left, which may be out of order, and the elements on the right, which are in final order. The outer loop exits whenever the inner loop didn't swap anything. If the collection was already sorted, then all we need do is one pass through the collection. But, if the collection happened to be in reverse order, we would have to run the outer loop a total of $N-1$ times because

the first element must find its way to the last place one position at a time. The average ending point for the inner loop is half-way through the collection so the number of compares we do in the worst case is $\binom{N}{2}$, i.e., all pairs.

Here is a graphical depiction of the process for a short array: the first pass is on the left, the second (final) pass is on the right:





The best case (when the collection happens to be sorted already) is $N - 1$ compares; the worst case is $\binom{N}{2}$ compares (all pairs); and the average case is in practice slightly less than for the worst case.

The fact that w (the grey element) always divides the unsorted partition from the sorted partition is an example of a "loop invariant" because it's valid at the same point in every iteration of the outer loop.

Bubble sort has almost no redeeming virtues as a practical sorting method. Even Senator Barack Obama knew that bubble sort was the "wrong way to go" in his 2007 Google "interview". But it is nevertheless instructive. And it does have one feature that cannot be claimed by most other popular sorting methods. Did you notice that we referred to the input as a *collection* of elements—without specifying whether it was an array or a list? Most sorting algorithms work efficiently only on arrays because they must do one or the other of the following:

(1) compare/swap non-neighboring elements.

(2) traverse the collection right-to-left instead of left-to-right.

These operations are quite awkward in a typical linked list that is defined left-to-right. In the pseudo-code (above) you can see that bubble sort will work for either an array or a linked list.

Swaps

Does it bother you that we're using the colloquial term *swap*? A more formal word would be *exchange*. But swap is just so much shorter and clearer. Now, it's time to introduce a term for the idea of a conditional swap. Let's call it a *coswap*, which you can think of either as a conditional swap or compare-and-maybe-swap.

An algorithm which is clever enough to do less work when the result has already been partially achieved is called *adaptive*. You might wonder why anyone would want to sort a collection that has already been sorted.

To help discuss that, we need to introduce another term: *inversion*. An inversion is simply when any two elements of a collection are not in their correct (desired) order.

The maximum possible number of inversions in a collection of size N is of course the number of pairs, $\binom{N}{2}$. As we've already noted above, that occurs when the collection is in *reverse* order. Note that the term "inversion" applies to both neighboring elements and non-neighboring elements. When a collection is already sorted, it has zero inversions.

If the minimum number of inversions is zero and the maximum number is $\binom{N}{2}$, then what's the average number? If a collection is in random order, then any pair has a 50% chance of being inverted. Thus, the average number of inversions (for a random collection) is $\frac{N(N-1)}{4}$.

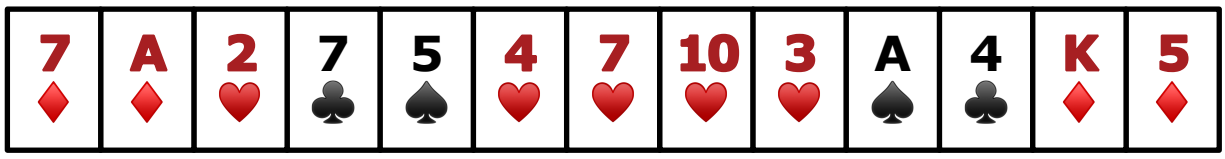
When the number of inversions in a collection of N elements is proportional to its length (rather than length squared), then we consider that collection to be *partially ordered*. Why would that ever happen? And how would we know about it.

Let's say you are a birdwatcher. Every day you go out and you note in your journal the names of the birds that you see. When you get home, you add those new numbers to your (ordered) lifetime list. Let's say you've already seen 100 birds. And today you saw eight more. How many inversions do you have, potentially? Today's list might have about 14, according to the formula. But each of those eight new birds might also be inverted with respect to the birds already in the list: that's 400 inversions, on average, for a total of 414. Assuming that the existing list is much longer than today's list, we can almost discount the smaller term and just think about the larger term, whose value is linear with respect to the length of the original list. If the 108-element list was in random order, we'd expect about 2900 inversions. Thus, the 108-element list is an example of a partially ordered list and will benefit from the use of an adaptive sort method.

This might seem like an unusual application. But database indexes get updated like this *all the time!* A transaction occurs which adds a few records to a table and the index must be updated. Surely you want to use an adaptive algorithm that's linear in the size of the existing table, rather than quadratic or linearithmic.

Insertion Sort

A sorting method such as bubble sort is called an elementary sort, because, in general, it works by coswapping *elements*. But there's another very important elementary sort that is familiar to most people, especially bridge (or whist) players. You are dealt 13 cards and most players like to sort them into suits, and perhaps sort them according to rank within the suits. For example,:



The technique used by bridge players is simply to scan the hand and look for a card that's out of place. Then they *insert* it into its proper place. The computer algorithm

that mimics this is called insertion sort.

Insertion sort is an improvement over bubble sort for random inputs because, in general, both the inner and outer loops are run, on average, $N/2$ times. By contrast, only the inner loop of bubble sort takes $N/2$ iterations: the outer loop requires N iterations (fewer if partially ordered).

Another way to put it is that, in insertion sort, each pass of the outer loop performs either zero or one additional comparisons. The difference comes from what terminates the inner loop: coming to the end of the elements (zero extra comparisons) or stopping because the moving element is in its proper place (one extra comparison). Therefore, in terms of the original number of inversions *X*, the number of comparisons for insertion sort is $\Omega\big(X + O(N-1)\big)$ and $\Omega(N-1)$. As you recall, $X = \frac{N(N-1)}{4}$ when the input is random.

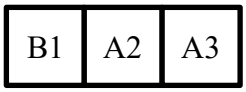
Java code: InsertionSortBasic.java.

Stable Sorting

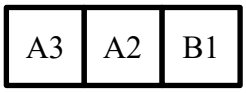
Thus far, our (conditional) swaps have always involved neighboring elements. This type of swap has an important property. Suppose that we have two neighboring elements that have equal keys. We wouldn't ever actually swap them since they're not out of order. That means that, if those elements are already ordered according to a different key, that original ordering will not be disturbed.

For example, when grading students from different sections, a learning management system (LMS) orders all students alphabetically. But you really want them to be grouped into sections, so you now order by the *section* key. You still want them alphabetical within each section, so you must use a stable sort. Spreadsheets know that you want to do this kind of thing, so spreadsheet sorts always are stable. And, because they use neighbor-swaps, both bubble sort and insertion sort are stable.

What is it about non-neighbor swaps that causes instability? Imagine three elements, and two keys (an alphabetical key and a numeric key):



They have been previously ordered by the numeric key and now you are sorting alphabetically using an unstable sort (uses non-neighbor swaps). When we compare the first and last elements, we see that they are out of order so we must swap them. This results in:



But now, A3 and A2 are no longer ordered numerically.

Selection Sort

Why would we ever use a non-neighbor swap? For efficiency. Neighbor swaps always fix exactly one inversion. That's why, if we start with the expected (quadratic) number of inversions in a random array, we must use a quadratic number of coswaps to sort it. That's usually a bad idea.

But non-neighbor swaps can fix multiple inversions. Consider the three elements discussed above in the previous section. And let's further imagine that the final ordering is actually what we wanted (reverse numerical followed by alphabetical). Then the original layout was in reverse order and had the expected $\frac{3*2}{2}$ (three) inversions: B1/A2, B1/A3, and A2/A3. The final layout is in order (zero inversions). Yet, we only performed one swap. Therefore, that swap was able to fix three inversions. In general, if an array of length N is in reverse order and we swap the outer two elements, we fix $2N-3$ inversions.

Now, let's imagine a different problem (one that does not often arise in software): it's trivially easy to compare two elements but very hard to swap two elements. For example, a delivery truck randomly deposits 10 packages on pallets (skids) in a line on the warehouse floor. Your manager asks you to position them in the same line according to their bar-codes. The packages are far too heavy for you to move so you need a fork-lift. You also have a scanner that can easily tell you which of two packages comes first. Naturally, to do this as efficiently as possible, you must minimize the number of swaps, while not being overly concerned with the number of comparisons.

Selection sort is the algorithm you need. You will have to pass over the elements of the array exactly nine ($N-1$) times. Each of these passes ends with a single non-neighbor swap operation. A pass involves finding the minimum (by bar-code) of the elements which are still out of order, that's to say those packages on the right of a (blue) marker that we place between the packages on the left which are in their final order, and those on the right that are still out of order. To begin, we place this blue marker to the left of the left-most package. Then, each pass—using only our scanner, and a second (green)

marker which is placed in front of the package to the right of the blue marker—requires comparing the value of the bar-code of the (green-marker) package with each package to its right. When we find a package with a "smaller" bar-code, we move the green marker to it. We then get into the fork-lift and swap the element immediately to the right of the blue marker with the package identified by the green marker. We conclude that pass by moving the blue marker one place to its right.

In our example, once we've completed nine passes, there's only one package left, which must already be in its proper place. Obviously, successive passes do fewer comparisons. Thus, we perform, in general, $\frac{1}{2}N\ (N-1)$ comparisons and $N-1$ swaps.

Because the swaps we do are not neighbor swaps, selection sort is not stable. But there is no better algorithm for this type of situation, where comparison is cheap but swapping is expensive.

There's one more relevant observation to make. Both insertion sort and selection sort *reduce* the problem by dividing their input into two partitions: one representing the result (on the left), and one representing work still to be done (the right). Each pass moves the boundary between the two partitions one place to the right. But in the case of selection sort, the "work" (finding the minimum remaining element) is done first and then it is added to the left partition. In the case of insertion sort, the next unknown element (from the right partition) is added to the left partition. Only then is the work—finding its proper place—carried out.

Optimal Sorting

For any problem, we should really try to determine the minimum number of operations that are required (the best case for the problem). Then we seek an optimal solution.

Let's start with the observation that there are $N!$ possible permutations of N elements of which only one is ordered. Suppose that we could randomly pick a pair of elements and compare them. According to whether they are inverted (or not) we can partition our set of candidate solutions into two: one set (the "good" set) includes the pair in order; the other does not. Now, we perform another random comparison, and divide the good set into two as before. What is the minimum number of such comparisons we must perform to end up with a set containing *only* the ordered permutation? The answer is $\log_2 N!$. But what is this number? According to Stirling's approximation, it is approximately $N\ \log_2 N$ (notated in the US as $N\ lg\ N$). Keep in mind that this is the theoretical minimum *possible* number of comparisons, if we make an ideal choice for each comparison.

Therefore, we can say that the problem of comparison-sorting an array of N elements requires at least $N\ lg\ N$ comparisons, i.e., $\Omega(N\ lg\ N)$. Any sorting algorithm whose order of growth is $O(N\ lg\ N)$ is considered optimal.

Suppose we simply perform $N\ lg\ N$ coswap operations on random pairs as our first attempt at an optimal solution. Approximately half of these will result in action and half will not. But, after each operation, we divide the solution space into the good half and

the bad half. The effect of this sort is to derive a partially ordered array. At the conclusion, then, we must apply insertion sort to fix the remaining inversions in linear time. In practice, because each pair tested is as likely to be inverted as not, the algorithm can be improved somewhat by performing $2\ N\ lg\ N$ coswaps.

The "random sort" described above doesn't do a very good job in practice because the pairs intelligently chosen (there will be repeated comparisons and other comparisons that don't contribute information). Note that random sort is not stable because it does non-neighbor swaps. It does sort in-place, though. However, it's not a good practical sorting method so don't look for it in your favorite sorting library.

Merge Sort

Can we find a good optimal solution, i.e., one which also requires something very close to the minimum number of comparisons? Indeed, we can—by using the technique of "wishful thinking," that's to say recursion, also known as divide-and-conquer (see Chapter 1).

First, we divide the input array of length N (this method only works well with arrays) into two partitions as evenly as possible. Next, we sort each partition recursively, resulting in two sorted sub-arrays. But what we want is the entire array sorted. For this, we must "merge" the two partitions. Unfortunately, try as you might, you cannot perform this operation in place. You will need another temporary array of length N in which to store the whole sorted array. Then this temporary array, known as the auxiliary array, must be copied back into the original array. This is a nuisance to say the least, but is unavoidable using this elegant sorting method, called merge sort.

However, we can use the same auxiliary array for the entire merge sort process, and we can also avoid one of the copy operations described above by exchanging array pointers at every level of recursion and also making the auxiliary array a clone when it is first allocated. Nevertheless, the use of the auxiliary array does slow merge sort down noticeably, simply because we must access twice as much memory as for other sorting algorithms.

How many compares are required? At each level the algorithm must do between $\frac{N}{2}$ and $N-1$ comparisons. We only need to do half the comparisons if the elements are already ordered. The number of levels, of course, is $lg\ N$. So, merge sort is indeed an optimal sorting algorithm, as we would expect from applying the Master theorem.

There are some other worthwhile optimizations, in addition to the trick of avoiding extra copies mentioned above. The first of these arises from an interesting, perhaps surprising, observation. $N\ lg\ N$ grows more slowly than ¼ N^2 of course. But if you compare their values where $N < 16$, i.e., the point where $N = 4 \lg N$, you will notice that insertion sort will do fewer compares than merge sort (in the random case). And, because it will not have to create extra stack frame entries for the recursion, it will do it more efficiently, too. So, at some threshold in the region of 16, it makes sense for merge sort to invoke insertion sort, instead of recursing into merge sort.

There is another optimization which makes sense if you think that the input array may be partially ordered. Consider the situation where the last element of the left-hand partition is smaller than the first element of the right-hand partition. In this case, we won't need to do any further comparisons—we simply copy all the elements to the destination. It's an example of taking out insurance. The premium is the extra test that you perform for every pair of partitions which, in many cases, will not benefit you. But, when it does pay off, it saves $\frac{m}{2} - 1$ comparisons (net) where m is the total number of elements in the two partitions.

One further observation regarding merge sort is that we don't do any swapping. All re-ordering is performed via copies.

An analysis of the required numbers of comparisons (k) reveals that:

- $k = O(n\,h - 2^h + 1)$
- $k = \Omega(\frac{n\,h}{2})$ (when the array is sorted)
- $k = \Omega(2^h - 1)$ (when the array is sorted, and the insurance test is used)

Where $h = \lceil \lg n \rceil$.

Of course, when n is a power of two, we can instead write:

- $k = O(n \lg n - n + 1)$
- $k = \Omega(\frac{n \lg n}{2})$ (when the array is sorted)
- $k = \Omega(n - 1)$ (when the array is sorted, and the insurance test is used)

> **Historical note**: Merge sort was introduced in 1945 by John von Neumann for the EDVAC. Von Neumann had to implement the recursive aspects of the method himself, resulting in 23 hand-written pages of code! The only alternatives for general comparison-based sorting were the "elementary" sorts, viz. insertion sort, bubble sort and selection sort. Von Neumann was generally considered by everyone who knew him as the smartest person they knew.

Quicksort

Does that optimization where we perform the "insurance" comparison suggest anything to you? What if we could somehow arrange for that condition always to hold true? But aren't we at the mercy of the elements as presented? What if—before performing the recursion—we swapped any element in the left-hand partition that belonged in the right-hand partition with an element that was similarly mislocated in the right-hand partition? Good idea, but it won't work without knowing the value of the median element. And to determine that every time we recursively invoked sort would be far too much work to be efficient. However, we could simply guess the median. Now, we might be able to do our swapping of out-of-place elements. There's another

snag, though. The median element will almost certainly not be in its proper place *before* sorting so, we will need to do our swapping based on where we expect our guessed median to end up. Because this element will be used for determining which elements are out of place—too "heavy" or too "light" for their actual position—we call this pseudo-median element the *pivot*.

Also note that, because in general we are guessing the pivot to be the median, it won't end up exactly in the middle. Indeed, we won't know its proper resting place until we have swapped all the misplaced elements.

But the benefit of this method is that, once all swapping has occurred, we will have a situation like the following, where v is the value of the pivot:

Now, we recursively sort the left-hand partition, as well as the right-hand partition and, when we're finished, no merging needs to take place. Not only does this eliminate the need for any more comparisons, but it also eliminates copying. Therefore, we no longer need that auxiliary array! This sorting method operates *in-place* and is called "quicksort".

> **Historical note**: Quicksort was developed in 1960 by Tony Hoare although he couldn't implement it right away because no languages of the time provided recursion (Algol 60 was the savior). I don't know if he thought about it the way I have described it, but it was a brilliant improvement to sorting at the time. The best alternatives for general comparison-based sorting were Shell sort, approximately $O(N^{\frac{3}{2}})$, and merge sort, linearithmic in comparisons but using twice the memory needed by the elements themselves. This may not sound like a big deal today, but if your memory was only 4k words, it was a huge limitation.

There is, however, one more issue to address. When we partitioned for merge sort, we partitioned the array as equally as possible. Can we still operate our recursive algorithm even though the partition sizes may not be the same? Yes, it will work just fine. However, the (exact) equal partitioning did result in the algorithm requiring just $n \lg n$ comparisons (see detailed analysis in sidebar X). Suppose that our guess for the median is so bad that we simply randomly choose any element as the pivot. We show (sidebar Y) that the number of comparisons required will be $2\, n \ln n$ which is approximately 39% more than for merge sort.

Randomly choosing the pivot has an even more serious problem. It is just possible that each time we choose the pivot, we pick the smallest element (or the largest). This will result in two partitions: one of zero size, the other of size $N - 1$. This will take N levels of recursion to successfully sort the array. That means, since each level will on average need to do $\frac{N}{2}$ comparisons, a total number of comparisons equal to $\frac{1}{2}N^2$ (i.e., no improvement over bubble sort, or selection sort). Therefore, it's important to answer $O(N^2)$ when asked "what is the worst-case complexity for quicksort?" But realistically,

the danger is far-fetched. There are several ways to avoid the problem in practice. The first, and perhaps least effective, method is to shuffle the array before starting quicksort or, equivalently, to pick a random element at each level of recursion (but the number of random elements used thus is linearithmic, not linear as in the initial shuffle). Either of these methods will reduce the odds of quadratic performance to about 1 in N^N.

A better method is to guess the median (as mentioned above) by, for example, taking the median of three elements, typically the first, last, and middle elements. Again, the likelihood of quadratic performance is vanishingly remote.

The state of the art is to take a leaf from the book of engineering redundancy. Just as an aircraft has multiple hydraulic systems, for example, quicksort can benefit from having multiple pivots. If it's unlikely that a single pivot will cause quadratic behavior, it's even less likely when two or more pivots are employed. It has been standard practice in recent years (in Java, for example) to use Yaroslavskiy's 2011 "dual-pivot quicksort."

> **Historical note**: Even when precautions were taken to avoid quadratic performance, quicksort was very slow when certain patterns were present in the input data. And it did not perform well when there were many duplicate elements (indeed, there was even a few incorrect implementations in the public domain, especially when duplicates were present). Various improvements were introduced over the years, including "three-way quicksort," and "dual-pivot quicksort."

Two other aspects of quicksort should be noted. Because the swap operations that perform the partitioning use general, non-neighbor swaps, quicksort is not stable. And, of course, quicksort is not adaptive. Even if we don't shuffle, we still must look at each element to see if it should be swapped.

As we noted regarding insertion and selection sorts, the order in which we performed the partitioning and the work can be significant. In merge sort, we partition and recurse first then merge (the work). In quicksort, we start with the rearrangement of elements (the work), then we partition according to the pivot and recurse.

Merge sort vs. Quicksort

The foregoing discussions of merge sort and quicksort have left out some important practical aspects. For example, quicksort typically performs more comparisons than merge sort, yet is generally considered to be faster. This is a good illustration of the fact that the number of instructions isn't the whole story. The time spent waiting for memory is often more important. Merge sort must deal with twice the memory. As a gross over-simplification, the time spent by merge sort waiting for cache pages to be refreshed is likely to be double the time spent by quicksort. For other, more subtle, differences, dual-pivot quicksort spends less time waiting for memory (i.e., is more "cache-friendly") than original quicksort.

You might have noticed that quicksort uses non-neighbor swaps (so is not stable) but that merge sort doesn't do any swapping at all. It *copies* elements. However, when we have two equal keys in the two different partitions, we can ensure that merge sort is stable simply by always choosing the element from the left-hand partition.

There's another significant difference between merge sort and quicksort which has to do with the complexity of individual comparisons. If we are comparing scalar values (e.g., integers, floating point values) directly, the time for a compare will necessarily be less than when a pointer must be followed to one (or more) scalar values. This is particularly because the destination of that pointer is likely to require a cache page refresh. In the Java world, we would call these two types of information primitives and objects, respectively. Because objects are always more expensive to compare, we want to minimize the number of comparisons—and thus we should use merge sort. But, for primitives, the fast comparisons allow us to use quicksort. And this is reflected in the default sorting methods that Java provides for objects and primitives.

One more factor which may make a small improvement in quicksort's favor is that all the comparisons in merge sort are array-element to array-element. Whereas, in quicksort, the comparisons are array-element to pivot (thus there is no chance of an additional cache page fault).

Timsort

There's another optimization that we could try for merge sort. Instead of using recursion, how about converting it to an iterative method? This is possible, even though merge sort, with two partitions to recurse on, would not normally be *tail-recursive* (chapter X). Nevertheless, because merge sort does not produce a *result* (instead, it mutates the given array), and is thus able to be iterated—the so-called "bottom-up" merge sort.

Unfortunately, bottom-up merge sort takes slightly longer than the top-down version. However, if the input array is partially ordered, it is possible to speed the process by looking for arbitrarily sized "natural" runs of elements in order. Similarly to its top-down cousin, merge sort, Timsort does not merge small runs, devolving to insertion sort instead. Thus, if a run is too short, it will be expanded using insertion sort until a threshold length has been achieved.

A further significant optimization in Timsort is that runs are merged intelligently. Typically, runs can be merged by using binary search to find where first/last elements of a run will end up in the merged run. This knowledge determines which of the elements must be copied into auxiliary storage. Typically, that number is significantly smaller then N.

> **Historical note**: Peter McIlroy introduced some of these ideas for merge sort in his 1993 paper "Optimistic Sorting and Information Theoretic Complexity". When Python was looking for a sort algorithm, Tim Peters made further optimizations (2002)

and Timsort was born. Today, Timsort is also the default sort method for *objects* in Java and various other languages.

Heap sort

In chapter 3, we learned about that remarkable abstract data structure called a binary heap, which is an ideal implementation of a priority queue. Suppose that we have an array to be sorted. We insert the elements into a minimum binary heap and then we successively delete them, putting the first (minimum) element in the first slot of the array and so on. We would have achieved an ordering of the array. Its time complexity would be $O(N\ lg\ N)$ while its memory complexity would be $O(N)$, i.e., accounting for the space required by the binary heap.

Because the sink/swim operations (the swaps) in the binary heap are not neighbor swaps, our sort method would not be stable. So, comparing it with merge sort, it has the same time and space complexity but is not stable. What use could it be?

But wait: the binary heap is itself an array. Could we perhaps avoid creating a new array for the binary heap? Let's try pretending that the input array is a maximum binary heap. If the array happened to be in reverse order, it would satisfy the "heap order" invariant. But, in the usual situation, it would not. But we already know how to fix a heap that is not in heap order: we apply the sink operation. Starting with the last sub-heap of two or more elements (which is exactly halfway through the array), and working backwards towards the root, we sink each sub-root as appropriate. By the time we have operated on the entire heap, it will be in heap order.

Incidentally, because the elements are not being inserted one-by-one into the binary heap (but are already present at the start), the number of comparisons required for this operation is $O(N)$ rather than $O(N\ lg\ N)$, as discussed in Chapter 3. However, the number of swaps remains $O(N\ lg\ N)$.

Once the binary heap has been thus constructed, the next step is to perform the delete-max operation for each of the N elements. The difference is that the result of each delete-max is not passed to a caller but is given back to the ownership of the original array. That's to say we swap the last element of the binary heap with the root element, perform the sink down of the new root, then we simply decrement the count. In this way, the largest element (the original root) ends up in the last place of the array, and so on.

What's the point? We already have an algorithm that is (on average) $\sim(N\ lg\ N)$, unstable, and in-place. It's called quicksort. And, it has a smaller coefficient than heapsort. But, if you recall, quicksort has a worst-case growth of $O(N^2)$. Heapsort has a natural use case, therefore, whenever quadratic performance cannot be tolerated, for example, in the kernel of the Unix operating system.

> **Historical Note**: Introsort (Musser, 1997) is a hybrid sorting algorithm which starts out using quicksort. If it finds that the recursion depth is getting out of hand (2 lg N), it switches to using heap sort. Both cut over to insertion sort when

a partition gets small. By the way, in a considerable amount of testing of Introsort, I have never, ever hit the threshold that invokes heap sort!

Shell sort

In the section on Optimal Sorting (above), we tried the idea of "random sort" as a pre-processor to insertion sort. The idea was to eliminate almost all inversions before the final pass of insertion sort. Any algorithm which can remove most inversions would be a suitable pre-processor (see Chapter 8, Advanced Sorts for more pre-processor designs). How about using insertion sort itself on sub-arrays of the array?

The problem we run into is that, however we might partition the array, neighbor swaps can only fix one inversion at a time.

But, suppose that, instead of the usual type of partitioning, we by *striding* through the array. Let's say that we want h such partitions. Partition 0 (colored red in the diagram below where $h = 3$ and $N = 16$) must therefore be made up of elements at indices: $0, h, 2h, 3h$, and so on up to $\frac{N}{h} h$. Partition 1 (green) would include indices: $1, h+1, 2h+1, 3h+1$, etc. Partition 2 (blue) would consist of the remaining elements. In general, the partitions won't all be the same length. Here, there are 5 reds, and three each of blue and green.

Now, let's consider an inversion between (red) elements at indices 0 and h. If we fix that inversion by swapping those elements, it's possible, that we could fix as many as $2h - 1$ inversions in the original array. In practice, the number of fixes will be more modest (and occasionally could even be negative). However, if the average number of fixes per swap is greater than one, this method can do an efficient job of pre-processing the array, albeit in an unstable manner.

So, the way each pass of this algorithm, known as Shellsort, works is that there are h independent insertion sorts—one for each of the partitions. The red sort will require, on average, $7\frac{1}{2}$ coswaps, the green and blue partitions will require an average of 5 coswaps each. At the close of the pass, each of the partitions will itself have zero inversions, yet approximately one-third of the initial number of inversions in the array, will remain. This is because we have not yet compared any of the red elements with green or blue elements.

How should we go about choosing h (also known as the "gap")? One simple method is to divide N by about three and then divide by three again for the next pass. The final pass will have $h = 1$, which is of course plain ordinary insertion sort. Successive gaps should be relatively prime, otherwise we will repeatedly compare the same pair of elements

The analysis of Shellsort is very complex and beyond the scope of this book. And there are many proposed gap sequences. One of the best is the Sedgewick (1986) scheme:

1, 5, 19, 41, 109, ... Many of the simpler gap sequences result in worst-case growth of $O(N^{\frac{3}{2}})$. However, Sedgewick's sequence grows at $O(N^{\frac{4}{3}})$ and the Pratt sequence $O(log^2 N)$. No good expression exists in general for the average growth rate.

When the input array is already ordered, each pass will do $\sim N$ compares. Thus, the total number of compares is pN where p is the number of passes. For the division-by-three scheme, $p = \log_3 N$.

Historical Note: Shellsort is mostly of historical interest today. But when D. H. Shell published it in 1959, the only alternatives were insertion sort and merge sort among the comparison sorts. Recall that, at the time, merge sort's almost fatal flaw was that it required double the memory—something in very short supply in those days.

