REASONING ABOUT EFFICIENCY

When we face a computational problem, of course we first want to find a correct algorithm for solving it. But given a choice between several correct algorithms, we would also like to be able to choose the fastest among them. Analyzing algorithms' performance is the subject of this chapter. After seeing the basic technique for theoretical analysis, called*big-O notation*, we'll look at its application to analyzing algorithms' speed for two significant problems: sorting arrays and counting primes. We'll conclude by looking at how the same concept can be applied to analyzing algorithms' memory usage.

Analyzing efficiency

Suppose, for example, that we want to write a program to determine *integer square roots*, which we'll define as the largest integer whose square does not exceed the query integer (in mathematical notation, $\lfloor sqrt(n) \rfloor$). For example, the integer square root for 30 is 5, since $5^2 \leq 30$ but $6^2 > 30$; for 80, it is 8; and for 100, it is 10.

The first technique one might imagine is this: We'll square consecutive numbers until we pass the query integer. Then we'll return one less than that.

```
public static int isqrtIncrement(int query) {
    int cur = 0;
    while(cur * cur <= query) cur++;
    return cur - 1;
}
```

Another technique is based on binary search: We'll maintain a range where we think the integer square root might be; and each iteration, we'll halve the range so that it still spans the integer square root, but one end of the new range is at the old range's midpoint.

```
public static int isqrtHalve(int query) {
    int low = 0;          // invariant: low * low <= query
    int high = query + 1; // and high * high > query
    while(high - low > 1) { // while range has >1 number
        int mid = (low + high) / 2;
        if(mid * mid <= query) low = mid;
        else                   high = mid;
    }
    return low;
}
```

While the technique as stated here is theoretically correct, this implementation has a major bug. Suppose we want to find the square root of 100,000, for example. This code will start by squaring half of that (50,000), and the result will be *overflow* — a number larger than an `int` can handle. This implementation is correct only for integers up to 92,680. By contrast, `isqrtIncrement` is correct for integers up to 2,147,395,599. For this chapter, however, we'll ignore such issues.

Which approach is faster? The answer may not be obvious looking at the code alone.

One way to determine this is to implement both algorithms and test. To do this, I wrote both and used them to find the sum of the integer square roots of the integers up to 90,000.

`isqrtIncrement`	326 ms
`isqrtHalve`	43 ms

While this answers the question at hand, the answer is not entirely satisfying: Other than that `isqrtHalve` is faster than than `isqrtIncrement`, what have we learned? All we have is a single result, without much feeling for *why* one is faster than the other. We would like a handle on the reason for the difference for several reasons:

- We'd like to be confident that the results obtained aren't peculiar to the particular computer on which we happened to run the tests.
- We'd like to be confident that the results aren't peculiar to the specific tests that we ran. Here, we should be pretty confident: The algorithms are both pretty simple, so we don't expect them to behave peculiarly in some cases. Moreover, the input is simple enough that we can test a very wide range of possible inputs.

  By contrast, if we wanted to analyze a program to sort an array, we'd need to try all different ways to arrange each array, on the off-chance that one of those ways happens to be a very bad case for the program. Testing such a program on such a large number of possible inputs would be prohibitive.

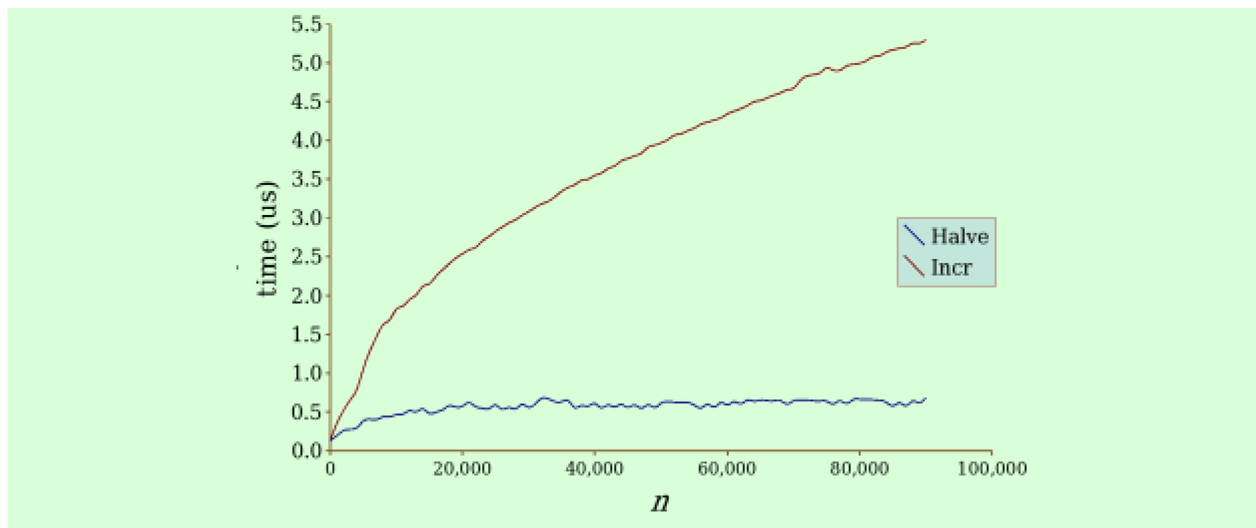
- We'd like to have a feel for the algorithms' weaknesses. If we can identify a bottleneck, then that gives us some insight where we should work hardest to improve performance — in effect, widening the bottleneck. The resulting algorithm may be faster than anything we thought of previously.

For all these reasons, we'd like a different way to think about the efficiency of a program.

4.1.1. Big-O notation

What we'll do is to imagine a graph of how the algorithm works relative to how big the input is. Actually, for just this example, we won't just imagine it: I'll *draw* it for you. You can see it in Figure 4.1. (Actually, this graph is not the true graph. The original graph based on the actual data was quite bumpy, and I smoothed it out to convey the methods' essential behavior. The bumps, as you see, did not entirely disappear.) I want you to appreciate all the time I've saved you: Measuring all those times to get that graph is quite a pain. Luckily, we won't have to do this manually again.

Figure 4.1: Comparing two square root algorithms experimentally.



When you look at the graph, you see something quite noticeable. Not only is `isqrtIncrement` much slower than `isqrtHalve` for large *n*, but the difference increases dramatically as *n* increases. In fact, the `isqrtHalve` curve looks almost flat at its large end, while `isqrtIncrement`'s curve continues to increase, though the rate of increase is slowly slowing as*n* grows.

In fact, what we're looking at here is the difference between an inverse parabola (i.e., $f(n) = a\ sqrt(n) + b$) and a logarithmic curve (i.e., $f(n) = a\ (\log n) + b$). With an inverse parabola, quadrupling *n* doubles the value (a $sqrt(4\ n) = 2$ a $sqrt(n)$), neglecting the *y*-intercept (*b*). You can see that `isqrtIncrement` follows this behavior: At $n = 20{,}000$, the time is 2.5 µs, whereas with $n = 80{,}000$, the time is double that, 5 µs. However, quadrupling *n* with `isqrtHalve` has hardly any effect at all on the time: At $n = 20{,}000$, the time is 0.58 µs, whereas at $n = 80{,}000$, the time is 0.66 µs. The curve for `isqrtHalve` starts out growing, but it quickly becomes very close to flat; this is characteristic of a logarithmic curve, which this happens to be (though with some noise added).

Any algorithm whose behavior is characterized by an inverse parabola will eventually be much slower than an algorithm characterized by a logarithmic curve. It doesn't matter what the coefficient *a* and the *y*-intercept *b* are for the two curves: These values affect the exact crossover

point, but no matter what, the inverse parabola will eventually surpass the logarithmic curve. (Well, it matters that a is positive. But it will be positive, since algorithms will be slower as n increases.)

We can actually visualize a whole hierarchy of different curves. For example, if we have another algorithm whose graph is linear, it will eventually raise above one characterized by an inverse parabola. (The inverse parabola, after all, slowly decelerates, whereas a line always grows at the same rate.) And a parabola, which is continually accelerating, would eventually go above a line.

We can tabulate this hierarchy.

slowest parabola	$f(n) = a\, n^2 + b$
line	$f(n) = a\, n + b$
inverse parabola	$f(n) = a\, sqrt(n) + b$
logarithmic curve	$f(n) = a\,(\log n) + b$

Computer scientists have a way of talking about these — and other — curves called big-O notation. Here, we say that the inverse parabola is $O(sqrt(n))$, whereas the logarithmic curve is $O(\log n)$. You can think of the capital O as hiding any constant multipliers, as well as any lower-order terms. Note that this means that not only is $a\, n^2 + b$ classified as$O(n^2)$: So is $a\, n^2 + b\, n + c$ and $a\, n^2 + b\, sqrt(n) + c$.

(You may be wondering how to pronounce this. Some people read $O(\log n)$ as big-O of log n; others say, order log n. Both are correct.)

This terminology generalizes to other functions, too. While we don't have an English term for a $O(\mathrm{n}^{1.5})$ curve, big-O notation gives us a way to talk about that function anyway. By the way, it ranks somewhere between $O(n)$ and $O(n^2)$.

In fact, the big-O bound is technically an *upper bound*. Thus, we can legally say that a line (which is $O(n)$) is $O(n^2)$ — or even that it is $O(n^n)$. This fact is important, because sometimes the nature of a curve is difficult to determine exactly, and so to get a result we may end up needing to cut some corners in our analysis by overestimating a bit. We'll see some examples of this later in this chapter, particularly in Section 4.3.

In talking about algorithms, we use this phrasing: `isqrtHalve` takes $O(\log n)$ time, while `isqrtIncrement` takes $O(sqrt(n))$ time. With the ranking of different functions internalized, we understand this as a fancy way of saying that `isqrtHalve` is much better than `isqrtIncrement` — at least for large n. (We're not so worried about small n here, because both algorithms are quite fast for small n anyway. It may be that `isqrtIncrement` is faster there, but since we're talking about fractions of a microsecond for small n in any case, distinguishing which is faster there isn't important.)

You might be wondering: What if we have two $O(sqrt(n))$ algorithms? Big-O analysis won't indicate which is faster, since big-O notation hides those constant coefficients. Unfortunately, there's no clean way around this: Those constant coefficients will depend on the particular computer for which we implement the algorithms, and they'll be very different for another computer. In fact, one of the $O(sqrt(n))$ algorithms may be faster on some computers, while the other is faster on other computers. But if we have a $O(sqrt(n))$ algorithm and a $O(\log n)$ algorithm, we know that regardless of the computer we choose, the $O(\log n)$ algorithm will be faster for large n.

Now, here's the important part: With just a bit of practice, we can look at an algorithm and quickly tell what its big-O bound is. We don't need to go through all the bother of programming it up, taking lots of measurements, drawing a graph, and then simply guessing what the curve looks like.

Let's take the `isqrtIncrement` method as a simple example.

```
public static int isqrtIncrement(int query) {
    int cur = 0;
    while(cur * cur <= query) cur++;
    return cur - 1;
}
```

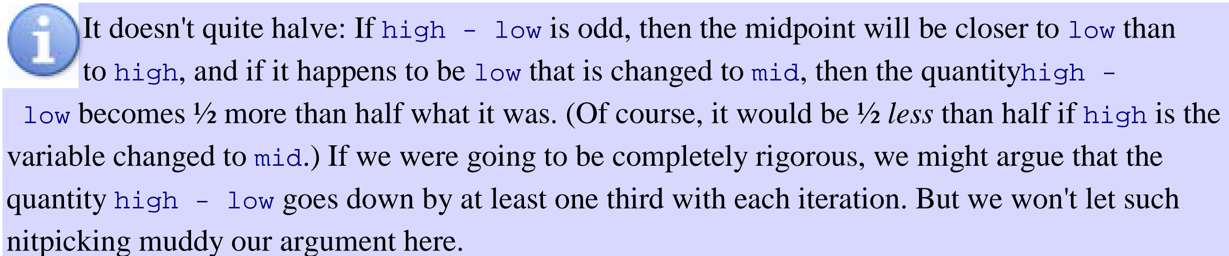
You can look at this and immediately see that we'll go through the loop for at most $sqrt(n) + 1$ iterations. Each iteration takes some constant a amount of time — some amount of time to test whether `cur * cur <= query`, and some other amount of time to increment `cur`. Thus, the loop will take $a\ (sqrt(n) + 1)$ time. In addition to this, there will be another constant b amount of time incurred once only: This will be the time to create the `cur` variable initialized to 0, to make the final test `cur * cur <= query` after the final iteration, and to subtract 1 from `cur` at the end. Thus, `isqrtIncrement` takes at most $a\ (sqrt(n) + 1) + b = a\ sqrt(n) + (a + b) = O(sqrt(n))$ time for some constants a and b whose values depend on the particular compiler and computer used.

We can apply similar reasoning to `isqrtHalve`.

```
public static int isqrtHalve(int query) {
    int low = 0;           // invariant: low * low <= query
    int high = query + 1; // and high * high > query
    while(high - low > 1) { // while range has >1 number
        int mid = (low + high) / 2;
        if(mid * mid <= query) low = mid;
        else                   high = mid;
```

```
    }
    return low;
}
```

The biggest difficulty here is determining the number of iterations for the loop. To get a handle on this, notice that the quantity `high - low`, which starts at `query` + 1, will halve each time.



It doesn't quite halve: If `high - low` is odd, then the midpoint will be closer to `low` than to `high`, and if it happens to be `low` that is changed to `mid`, then the quantity `high - low` becomes ½ more than half what it was. (Of course, it would be ½ *less* than half if `high` is the variable changed to `mid`.) If we were going to be completely rigorous, we might argue that the quantity `high - low` goes down by at least one third with each iteration. But we won't let such nitpicking muddy our argument here.

We can tabulate, then, where `high - low` will be after each of the first few iterations.

iterations	**`high - low`**
0	`query` + 1
1	(`query` + 1) / 2
2	(`query` + 1) / 4
3	(`query` + 1) / 8
4	(`query` + 1) / 16

In general, after k iterations, `high - low` will be at (`query` + 1) / 2^k. The method stops once this quantity reaches 1, which leads to the below equation; we have only to solve that equation for k to arrive at the number of iterations.

$(\texttt{query} + 1) / 2^k = 1$
$\texttt{query} + 1 = 2^k$
$\log_2 (\texttt{query} + 1) = k$

Thus, the loop will stop after $\log_2$ (`query` + 1) iterations. In general, whenever we have a loop that stops once a quantity reaches a constant, and that quantity starts at x and decreases by a constant percentage with each iteration, the loop will stop after $O(\log x)$ iterations.

By the way, you might have noticed that the logarithm's base disappeared. We write $O(\log n)$ rather than $O(\log_2 n)$. This is because the logarithm's base, when it is constant, is unimportant: The well-known logarithmic identity $\log_a x = (\log_b x) / (\log_b a)$ says that changing the logarithm's base from one constant a to another constant b simply changes the coefficient by $1 / \log_b a$. Since big-O notation hides constant coefficients anyway, the base is irrelevant, and we might as well omit it.

Big-O analysis allows us to approximate and compare algorithms' performance without actually implementing them. While it only provides the roughest estimate of running time, that estimate works out well when we are concerned with performance for very large inputs.

4.1.2. Multiple loops

To become more familiar with using big-O notation, we need to examine more examples. The examples above were fairly simple, involving only a single loop. Often, however, algorithms involve multiple loops. Consider the following example intended to test whether all of the integers in an array are unique.

```
public static boolean areAllUnique(int[] a) {
    for(int i = 0; i < a.length; i++) {
        for(int j = i + 1; j < a.length; j++) {
            if(a[i] == a[j]) return false;
        }
    }
    return true;
}
```

For each integer in this array, this method looks at all the integers following that one (note that `j` starts at `i + 1`) to see if any match it. If the inner loop finds a match, then it can return *false*; but if there are none found, then the outer loop continues to the next integer.

Note that in this example, the input is an array, not a number. When we want to analyze the speed of operations on an array, it is typically expressed in terms of the array's length. We'll use n to refer to the length.

One way to analyze the speed of methods using multiple loops is to start at the innermost level and move outwards. In this case, we can determine that the loop over `j` involves at most n iterations for each `i`, where n is the array's length, and each iteration of that loops takes $O(1)$ time. (Often, the number of iterations is much less than n, but remember that we are willing to overestimate with big-O notation. In this case, as it happens, the average number of iterations is $n / 2$, and the constant multiplier ½ won't show up in the big-O bound anyway.) Thus, the inner loop (over `j`) takes a total of $O(n)$ time it is executed. Executing this loop constitutes the whole of each iteration of the outer loop (over `i`), which involves at most n iterations. Thus, the outer loop will take at most $n \cdot O(n) = O(n^2)$ time. The final **`return`** statement will take an additional $O(1)$ time, so the total time taken is $O(n^2) + O(1) = O(n^2)$.

Multiple loops don't always mean that we multiply the number of iterations per loop, though. Consider the following simple method which finds the average of an array and then replaces each number that exceeds the average with the average instead.

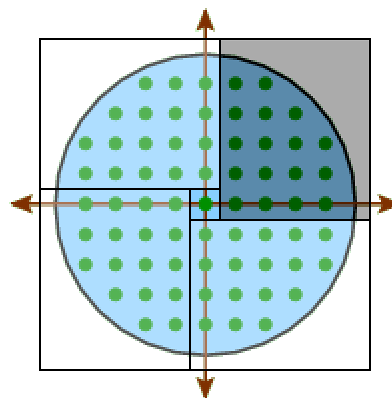
```
public static void cutLargeNumbers(int[] a) {
    int total = 0;
    for(int i = 0; i < a.length; i++) total += a[i];
    int average = total / a.length;
    for(int j = 0; j < a.length; j++) {
        if(a[j] > average) a[j] = average;
    }
}
```

In this case, the loops are not nested, so the amount of time is the time for the first loop, followed by the time for the second loop. Here, the first loop will take $O(n)$ time (since each of the n iterations takes $O(1)$ time), and the second loop will take $O(n)$ time (since each of the iterations also takes $O(1)$ time). The pieces in between each take $O(1)$ time, so the total is $O(1 + n + 1 + n) = O(n)$.

This type of analysis restricts itself to two questions: What is the maximum amount of time that each iteration of the loop can take? And what is the maximum number of iterations for the loop? With these two values, then we can multiply them together to get the total time for the loop. If we have one loop nested within another, then this leads to multiplication of the numbers of iterations; and if one loop is entirely after another, this leads to addition of their running times.

This technique is sound, but sometimes it ends up overcounting, particularly if the *maximum* time per iteration is much more than the *average* time per iteration.

Consider, for example, the problem of computing the number of *lattice points* within a circle — that is, the number of points whose coordinates are both integers. A circle of radius 5, for example, contains 69 lattice points, as drawn below.



We will count the lattice points by first determining the number within the upper quadrant, counting those on the x-axis but not on the y-axis — i.e., those in the darkened box above. We can then multiply this by 4, to take care of the other quadrants (the lightened boxes), and then we can add 1 to account for the origin. Here is the code.

```java
// Count integer-coordinate points in radius-r circle
public static int countLatticePoints(int r) {
    int count = 0;
    int y = 0;
    for(int x = r; x > 0; x--) {
        while(x*x + y*y < r*r) y++;
        count += y;
    }
    return 4 * count + 1;
}
```

A simple analysis of this code is that the inner loop will always iterate at most r times before $x^2 + y^2$ exceeds r^2. Thus, the inner loop takes $O(r)$ time. The outer loop has $r$iterations, giving a total of $O(r^2)$ time.

While technically correct, $O(r^2)$ is not the best possible bound for this code. In particular, since y begins at 0 at the method's beginning and never decreases, the *total* number of iterations of the inner loop — across all iterations of the outer loop — is r. Thus, the total amount of time spent on the inner loop during the method is $O(r)$; and the total time spent on the outer loop, excepting the inner loop, is $O(1)$ per iteration, over r iterations for a total of $O(r)$. Thus, the total time for the loop is $O(r)$.

Source : http://www.toves.org/books/data/ch04-bigo/index.html