Chapter 5

Efficiency and Algorithms

5.1 The big picture

We can now write code that does quite a lot, but sometimes it's not enough for a function to give the correct output. For example, let's look back at our functions that compute the Fibonacci numbers. In subsection 1.7.5 we gave the following code:

```
def iter_fib(n):
    prev, curr = 0, 1
    i = 1
    while i < n:
        prev, curr = curr, prev + curr
        i += 1
    return curr
```

You could also do this recursively. That code would be very simple:

```
def rec_fib(n):
    if n == 1 or n == 2:
        return 1
    else:
        return rec_fib(n-1) + rec_fib(n-2)
```

Both of these are, in terms of style, perfectly good ways to write the `fib` function. Both are simple and readable. But if you run them, you will quickly notice a big difference. The iterative version will run pretty quickly. You can run `iter_fib(100)` and get an answer in the blink of an eye. The recursive version, though, will take longer. For very small inputs (say 10 or 15) it runs pretty quickly, but as you increase the input value it quickly becomes noticeably slow. By the time you get to 50 or more you are likely to see *very* long run times, to the point where you have to just interrupt the program — it will not finish in a remotely practical amount of time.

Our goal in this chapter is to learn how to write code that runs quickly. (Or at least to get a start on that — a huge portion of computer science is devoted to making various things run more quickly, and we're only able to get to the very beginning ideas here.) Before we can even talk about how to make things run quickly, though, we need to be able to talk about running times in a useful way.

The most obvious way to talk about running times is to use actual time. I could note, for example, that on my computer running `rec_fib(20)` took 2.2 milliseconds, while running `rec_fib(40)` took 38 seconds. But how useful is that information if you're trying to evaluate the efficiency of `rec_fib` as a piece of code? It's certainly something, and it gives you a general idea, but it's far from a complete picture.

One problem is that you know nothing about my computer, so you don't know if I'm using an old smartphone that runs very slowly, or a state of the art supercomputer. Even if I tell you that it's a "normal" personal desktop computer, you're missing the level of detail needed to get a good picture of how the code will run on other machines. You would need to know the precise hardware. You would need to know what else I was running at the time that might have been competing for the attention of my processor. Even if you knew that, it would only give you an idea of what the running time was for two particular inputs. Maybe 20 is a particular easy case, or maybe 40 is a particularly hard one.

Now precise timings like those given above really do matter a lot. If you're trying to optimize a router that your company will be selling, you know exactly what sort of computations it's going to be doing and what the hardware will be and you care about getting squeezing every bit of efficiency out of that machine. But here we are going to look at a bigger picture. We want to ask "What is the fastest way to compute Fibonacci numbers?"

What we care about when answering this question is the code itself. We want to know what method will require the least amount of computation. We will refer to a particular method as an *algorithm*. An algorithm is simply a fully-specified process for doing something. A recipe for baking a cake is an algorithm. So are directions to the airport. The important thing about an algorithm is that it gives someone or something carrying out the algorithm an exact explanation of what to do at each step.

A piece of code specifies in algorithm, but computer scientists generally refer to a particular piece of code as an *implementation* of a particular algorithm, and use the word "algorithm" to mean something slightly more general. For example, consider the following code:

```
def rec_fib2(n):
    if n  > 0 and  n < 3:
        return 1
    else:
        a = rec_fib2(n-1)
        b = rec_fib2(n-2)
        return  a + b
```

This is a different piece of code, but you can see that it is basically carrying out the same process as our `rec_fib` definition above. Computer scientists would generally say that `rec_fib2` is a different implementation of the same algorithm implemented in `rec_fib`, whereas `iter_fib` is a fundamentally different algorithm. This distinction is a bit vague — certainly `rec_fib` and `rec_fib2` are not carrying out an absolutely identical computation — but they're close enough that the differences matter very little compared to the difference between `rec_fib` and `iter_fib`.

For almost every interesting question we might want a program to answer, the problem will get harder as the inputs get bigger, and we care a lot more about what happens with big inputs than what happens with small ones. Small inputs will generally be very easy for any algorithm. But efficient algorithms will be more able to handle large inputs in a

reasonable amount of time. How the efficiency of an algorithm behaves *as the input gets bigger* is what we refer to as *asymptotic* running time, and we'll deal with that in detail in the next section.

5.2 Asymptotic run time and Θ

Let's think about how long it takes `iter_fib(n)` to run. Whenever it's run, some initial work is done, assigning initial values to `prev`, `curr`, and `i`. Then there is some work that is done `n` times (evaluating `i<n`, adding `prev+curr`, and updating `prev`, `curr`, and `i`). Now, we don't know how long that initial work takes — it will depend on the particular machine running it. So let's call that time a. We also don't know how long the updating step takes, so let's call that b. Since the updating work happens n times, our total running time is $a + bn$. We don't know any exact numbers, but we do know that it grows *linearly* as a function of n. As n gets very big, the initial a-time work will be negligible, and the running time will be roughly proportional to n.

Now let's compare this to how `rec_fib` works. The initial call to `rec_fib(n)` does a certain amount of work itself, and it also makes calls to `rec_fib(n-1)` and `rec_fib(n-2)`. Each of those two calls will eventually make two calls of their own. Then each of those will make two calls and so forth until some of these calls start to hit base cases. Below is a tree that shows what this would look like for `rec_fib(6)`.

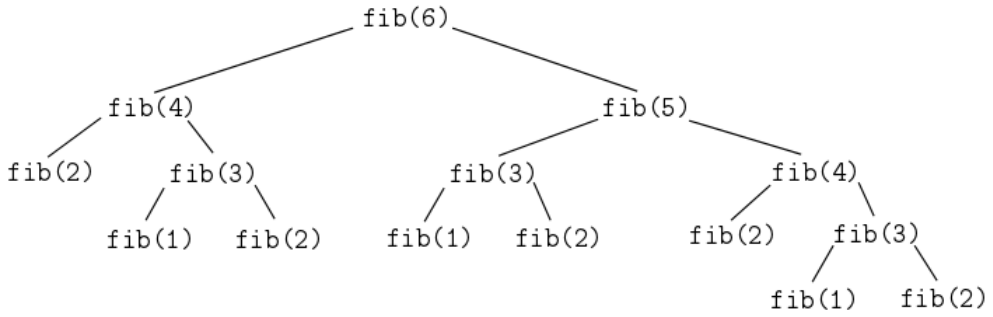

Figure 5.1: The recursive calls needed to evaluate `rec_fib(6)`. (We use the old `fib` function name to fit the tree more neatly.)

Because there's a constant amount of work per call (excluding the new recursive calls), to figure out how fast `rec_fib(n)` is we just have to figure out how many recursive calls will end up being made. Now, figuring out the size of this tree is complicated because some branches go down farther than others. But the first branch to end (on the far left in our figure) will be the one that has the input being reduced by two each call. Instead of trying to compute the exact number of calls needed to evaluate `rec_fib(n)`, let's just put a lower bound on it. All branches in the tree will go down at least $n/2$ steps, so let's ignore everything below that. The number of calls at each layer of the tree doubles, so we're looking at roughly $2^{n/2}$ calls. Each one takes some amount of time, which we can call c, so our overall running time is at least $2^{n/2}c$. This is an *exponential* function of the input size.

Just to get an idea of how big a difference this can make, let's say that the amount of work represented by the constants a, b, and c takes 1 millisecond. The Table 5.1 below shows how long each algorithm would take for each input size.

input $= n$	2	10	40	60	100
`iter_fib` time $= an + b$	3 ms	11 ms	41 ms	61 ms	101 ms
`rec_fib` time $= 2^{n/2}c$	2 ms	32 ms	17.5 minutes	12.4 days	35,678 years

Table 5.1: The running times of `iter_fib` and `rec_fib` on various inputs. The constants a, b and c are all set to 1 millisecond (ms).

You can see that `iter_fib` begins at roughly the same running time as `rec_fib`, but as n increases `iter_fib` stays reasonable while `rec_fib` quickly begins to take absolutely ridiculous amounts of time. The crucial thing to realize, though, is that this isn't about these two particular functions. Say we slowed `iter_fib` down by a factor of 1000, making a and b equal to a second instead of a millisecond. And say we sped `rec_fib` up by a factor of 1000, setting c equal to a microsecond instead of a millisecond. Table 5.2 shows the new running times.

input $= n$	2	10	40	60	100
`iter_fib` time $= an + b$	3 seconds	11 seconds	41 seconds	1 minute	1.7 minutes
`rec_fib` time $= 2^{n/2}c$	.002 ms	.032 ms	1 second	17.9 minutes	35.7 years

Table 5.2: The running times of `iter_fib` and `rec_fib` on various inputs. The constants a, and b are set to 1 second, while c is set to one microsecond.

By making `rec_fib` 1000 times faster and `iter_fib` 1000 times slower, what we've done is the equivalent of running `rec_fib` on a machine that is a *million* times faster than the machine that runs `iter_fib`. And for low inputs, that matters. Even at $n = 40$, `rec_fib` is now meaningfully faster. But by $n = 60$ `rec_fib` is much slower, and by $n = 100$ `rec_fib` is back to essentially never finishing. (Good luck finding a computer than can run for 35 years without breaking.) The point here is that regardless of the constants, basically *any* function with a linear runtime will be preferable to *any* function with an exponential runtime.

We want to be able to talk about the running time of a program at this high level, ignoring the details of what constants it has or how it behaves on small inputs and focusing instead on its behavior as the input gets large. We want to group functions into families that are all essentially the same. For example, we want running times represented by $f(n) = 2n + 7$ and $g(n) = 15n - 6$ to both be in the class of *linear functions*. The reason we think of f and g as equivalent is that which is faster could be changed by running one or the other on a faster machine. If g is run on a faster machine, it will be faster even for very high values of n. (Compare this to an exponential function, where no matter how fast we make its machine, for large enough n the linear function will still be better.)

We use the notation $\Theta(f)$ to represent the family of functions that "equivalent" to f. Another function g is in the family if for big enough n, $g(n)$ can be consistently made higher or lower than $f(n)$ by multiplying it by some constant. More formally:

Definition 1 *We say that a function g is in $\Theta(f)$ if there exist constants k_1, k_2, and m such that*

$$k_1 g(n) \leq f(n) \leq k_2 g(n)$$

whenever $n \geq m$.

For example, if we pick $f(n) = 2n + 7$ and $g(n) = 15n - 6$ as above, we can show that $f \in \Theta(g)$. To do this, we need to show a k_1 such that

$$k_1 g(n) \leq f(n)$$

for large n. This of course is the same as

$$k_1(15n - 6) \leq 2n + 7.$$

It's fine to pick a value of k_1 that is lower than we need, so let's pick $k_1 = .1$. Then we have for large n. This of course is the same as

$$1.5n - .6 \leq 2n + 7$$

which simplifies to

$$-1.6 \leq .5n.$$

This is true for all positive n, so our choice of k_1 works.

Now we also need to show the other inequality. Name, we need to show that there's a k_2 such that

$$f(n) \leq k_2 g(n)$$

or equivalently

$$2n + 7 \leq k_2(15n - 6).$$

Here we can just pick $k_2 = 1/3$ and simplify, getting

$$2n + 7 \leq 1/3(15n - 6),$$

$$2n + 7 \leq 5n - 2,$$

$$9 \leq 3n.$$

This last inequality isn't always true, but that's ok. We just need it to be true when $n \geq m$, and we get to pick m. The first inequality worked with $m = 0$, but this one requires m to be 3 or greater. So picking $m = 3$ makes both inequalities true for all $n \geq m$ and we're done.

If you think about it for a little bit, you should be able to convince yourself that if f and g are *any* linear functions, we will have $f \in \Theta(g)$ and $g \in \Theta(f)$. We generally represent this family as $\Theta(n)$, just because n is a simpler function than $2n + 7$.

If we were to look at quadratic functions instead, written as $an^2 + bn + c$, you would find that regardless of the constants, the function is always in $\Theta(n^2)$. In general, when you add multiple terms, the slower-growing terms (like bn or c) are irrelevant, and the family of the function (called its *order of growth*) is determined simply by the fastest-growing term (in this case an^2).

Frequently we will find running times that are polynomials, and only the degree of the polynomial matters. Algorithms that run in $\Theta(n)$ (linear) time are usually pretty fast, whereas $\Theta(n^2)$ (quadratic) or $\Theta(n^3)$ (cubic) time algorithms are substantially slower. Things that take exponential time are slower than *any* polynomial-time algorithm, and they are generally infeasible for all but the smallest of inputs.

Connection to limits If you're familar with the definition of limits, you'll see that our definition of Θ looks very familar. It's similar to a limit as n approaches infinity, with m serving the role of δ and k_1 and k_2 serving the role of ϵ. In fact, you can show that if $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ is a nonzero constant then $f \in \Theta(g)$. (The implication doesn't go the other way, but only because of weird cases. For example, the value of $\frac{f(n)}{g(n)}$ could oscillate between 2 and 3. In that case there would be no limit but nevertheless $f \in \Theta(g)$.)

Other resources In the brief introduction to efficiency we're doing here, we are only going to talk about running time. However, there are other resources beyond time that might be concerns for an algorithm. One important resource is *space*, meaning the amount of memory needed for storage while an algorithm is run. You can also talk about something using quadratic space, for example, and decreasing the needed space for an algorithm might be important in some situations. We might also care about *communication*, meaning how much data needed to be sent over a network connection to carry out some interactive computation. These things area all measured using the same asymptotic notation that we're using here for time.

5.3 Search

Calculating Fibonacci numbers is a good motivating example, but it's not something computers have to do very frequently. Now that we have an understanding of how to talk about efficiency, let's look at some more commonly used functions. We're going to start with search. The search problem takes as input a list and a value, and it outputs `True` if the value is contained in the list and `False` otherwise. This replicates exactly the built-in functionality of the `in` operator.

The most straightforward thing to do is to scan the list, checking each element. We can do that with the following code:

```
def search(alist, val):
    i = 0
    while i < len(alist):
        if alist[i] == val:
            return True
        i += 1
    return False
```

How long does this take? Well, before we can answer that question we always have to define how we're measuring the size of the input. Here we'll set n equal to the number of elements in the list. Given that, it's pretty easy to see that this is a $\Theta(n)$-time function. There's a constant amount of setup work (meaning that it doesn't depend on the size of the input), and then there's a constant amount of work that has to be done for each value of `i`, until either the value is found or we reach the end of the list. This of course is the *worst-case* running time, and that's what we'll care most about. If you knew a probability distribution that represented the distribution of the inputs, you could talk about average-case running time, but since we're often talking about writing a function that could be used in lots of situations with lots of different input distributions, computer scientists focus more on worst-case running time.

This is the same algorithm that Python runs automatically if you write `val in alist` in your code. It helps to know how these things work. Not every simple, one-line operation in Python is constant time. Particularly when lists are involved, lots of operations are actually much longer, often $\Theta(n)$, where n is the length of the list.

Now, $\Theta(n)$ isn't *too* bad, but it would certainly be good if we could do this faster. In general, we can't — if you don't look at every element of the list, you can't return `False` without the possibility of being wrong. However, there is a special case where we can go faster. Say the list is sorted.[1] Then we can sometimes know the answer without looking at anything. We could, for example, do the following:

```
def search(alist, val):
    i = 0
    while i < len(alist) and alist[i] < val:
        if alist[i] == val:
            return True
        i += 1
    return False
```

This code works as before, but if it sees a value in the list that is greater than `val` it will stop early. (Once one number is seen that is too high, the remaining portion of the list must be even higher and therefore can't include `val`.) Is this faster? Well, for some inputs where it would otherwise have to scan the whole list, it will now return `False` earlier. This could, depending on how often you see what inputs, save some time. But it could also slow things down because an extra comparison needs to be done at each step. But at the higher level where we think about worst-case asymptotic running time, nothing has changed. The constant amount of work we have to do for each value of `i` is a slightly higher constant now because of the additional comparison, but it's still constant. The number of times we'll have to do that work might be lower for some inputs, but in the worst case it's still n. (In the worst case, `val` is not in `alist` but is also higher than everything in `alist`.) So the running time is still $\Theta(n)$.

So what else can we do? Well, one option is to do what computer scientists call *binary search*. Instead of starting at the beginning, we'll first check the *middle* value in the list. If that is the value we're looking for, then we're done. If it's not, we don't just eliminate that value as a possibility. We can see if the value we're looking for is greater than or less than the middle value. If it's less, we know it must (if it's in the list at all) be in the first half of the list. If it's greater, it must be in the second half. So with one comparison we eliminate half of the list.

And we can continue this. We next check the middle element of whichever half of the list is left, and then we eliminate half of the remaining elements. We can keep repeating this, removing half of the list (plus the midpoint) each time, until we either find the element or we've removed the entire list from consideration.

Code for binary search is below. Remember that this only works if the list is sorted. The variables `first` and `last` keep track of the beginning ending elements of the portion of the list that could still possibly contain the desired value. If the value is not found, eventually `first` will be greater than `last` and the function will return `False`.

[1] What "sorted" means might depend on what's in the list — numbers might be sorted lowest to highest, strings might be in alphabetical order, etc. All we mean by "sorted" is that there's some way to compare items, and that the order they're in follows these comparisons. It's probably easiest to think of numbers as the standard example, but pretty much anything can be sorted.

```
def bin_search(alist, val):
    first = 0
    last = len(alist)-1
    while first <= last:
        mid = (first + last) // 2
        if alist[mid] == val:
            return True
        elif alist[mid] > val
            last = mid - 1
        else:
            first = mid + 1
    return False
```

So how fast is binary search? Well, each iteration of the loop takes the same amount of time, so we just have to figure out how many iterations there will be. We'll simplify things a bit by ignoring the rounding and the removing of the middle element and just say that the portion of the list under consideration halves each time. If we let t be the number of iterations, that means that the function will run until $n/2^t < 1$, or equivalently until $n < 2^t$. What power of 2 is the first greater than n? That's simply $\log_2(n)$ (rounded up).

The running time of `bin_search` is $\Theta(\log(n))$, *logarithmic*. Logarithmic growth is really the best one can realistically get (except constant time is rare for difficult problems). It is faster than any algorithm with a polynomial running time, even faster than $\Theta(\sqrt{n})$.

Note that we just say $\Theta(\log(n))$, dropping the base on the logarithm. That's because the base on the logarithm doesn't matter. Say we had functions whose running times were $\log_3(n)$ and $\log_{10}(n)$. You might remember the change of base identity for logarithms. It states that

$$\log_3(n) = \frac{\log_{10}(n)}{\log_{10}(3)}.$$

But $\log_{10}(3)$ is a constant, so these two running times are a constant multiple of each other, meaning they are equivalent in Θ-notation. The same holds for any choice of two bases.