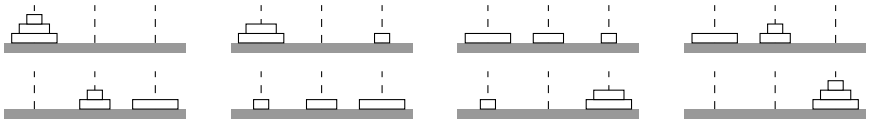
Recurrences are formulas that describe the value of a function of positive integers at an input in terms of the value of the same function at smaller inputs. We will see examples of problems in which recurrences come about and show how to solve them.

1 Towers of Hanoi

You have three posts and a stack of n disks of different sizes, stacked up from largest to smallest, on the first post.

The objective is to move the stack on to the third post, one disk at a time, so that a larger disk is never placed on top of a smaller one. Here is a way to do it for three disks:



What about four disks? You can try doing some experiments, but you might find the process quite involved. Instead, let us think inductively and use our solution for 3 disks as a "subroutine":

More generally, suppose we have solved the Towers of Hanoi problem for n disks. Here is how to solve it for $n+1$ disks: Ignore the largest disk and apply the solution for n disks to move the remaining tower to the middle pole. Then move the largest disk to the rightmost pole. Ignore the largest disk again and the apply the solution for n disks to move the remaining tower to the rightmost pole.

Let $T(n)$ be the number of moves we performed to move n disks. Then $T(n)$ satisfies the equation

$$T(n+1) = 2T(n) + 1. \tag{1}$$

Here, each of the two $T(n)$s accounts for the steps taken in each of the inductive moves applied to the tower of n smaller blocks, and the extra one is for the move of the largest block from the left pole to the right pole.

This type of equation is a *recurrence*. If we know $T(1)$, it allows us to calculate $T(2)$, $T(3)$, and so on. In our case, $T(1) = 1$ since a one block tower can be rearranged in one move.

2 Solving recurrences

One objective is to understand how a recurrence like (1) behaves for large values of n. The best thing to do is to get a closed-form formula for $T(n)$, if we can.

To do this I recommend you work backwards. From equation (1) we get

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

You start to spot a pattern: If we continue this process for enough steps until we get a $T(1)$ on the right hand side – this is $n-1$ steps – we get

$$T(n) = 2^{n-1}T(1) + (2^{n-2} + \cdots + 2 + 1).$$

Since $T(1) = 1$, it seems that

$$T(n) = 2^{n-1} + 2^{n-2} + \cdots + 2 + 1 = 2^n - 1$$

using the formula for geometric series from last lecture.

We can indeed verify that this guess is correct:

Theorem 1. *The assignment $T(n) = 2^n - 1$ satisfies equation (1) for all $n \geq 1$.*

Proof. Assume $T(n) = 2^n - 1$ and $n \geq 1$. Then

$$2T(n) + 1 = 2 \cdot (2^n - 1) + 1 = 2^{n+1} - 2 + 1 = 2^{n+1} - 1 = T(n+1). \qquad \square$$

Let's do another example. In lecture 10 we will talk about a sequence of graphs $B_1, B_2, \ldots$ called the Beneš networks that are defined recursively. Graph B_1 has 4 edges. Graph B_n is constructed from two copies of B_{n-1} plus an additional $2 \cdot 2^n$ edges. The number of edges $E(n)$ of B_n is given by the recurrence

$$\begin{aligned} E(n) &= 2E(n-1) + 2^{n+2} \\ E(1) &= 4. \end{aligned} \tag{2}$$

Let us try to solve this recurrence by working backwards:

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

Continuing this reasoning for $n-1$ steps we obtain the guess

$$E(n) = 2^{n-1}E(1) + (n-1) \cdot 2^{n+2} = 2^{n-1} \cdot 4 + (n-1) \cdot 2^{n+2} = n \cdot 2^{n+2} - 2^{n+1}.$$

We can now verify that our guess is correct:

Theorem 2. *For all $n \geq 1$, $E(n) = n \cdot 2^{n+2} - 2^{n+1}$.*

Proof. We prove the theorem by induction on n. For the base case, $1 \cdot 2^3 - 2^2 = E(1) = 4$ as desired. Now assume $E(n) = n \cdot 2^{n+2} - 2^{n+1}$. Then

$$E(n+1) = 2 \cdot (n \cdot 2^{n+2} - 2^{n+1}) + 2^{n+3} = n2^{n+3} + 2^{n+2} = (n+1)2^{n+3} - 2^{n+2}$$

as desired. $\square$

3 Merge sort

Merge sort is the following procedure for sorting a sequence of n numbers in nondecreasing order:

Input: A sequence of n numbers.
Merge Sort Procedure:
If $n = 1$, do nothing. Otherwise,
Recursively sort the left half of the sequence.
Recursively sort the right half of the sequence.
Merge the two sorted sequences in increasing order.

For example, if the input sequence is

10 7 15 3 6 8 1 9

the sequence is split in the first half 10 7 15 3 and the second half 6 8 1 9. Each half is sorted recursively to obtain

3 7 10 15 and 1 6 8 9

Then the two sequences are merged. To merge the first and second half, we compare the leftmost numbers in both sequences, take out the smaller of the two and write it as the next term in the output sequence until both halves become empty.

first half	second half	output
3 7 10 15	**1** 6 8 9	
3 7 10 15	6 8 9	1
7 10 15	**6** 8 9	1 3
7 10 15	8 9	1 3 6
10 15	**8** 9	1 3 6 7
10 15	**9**	1 3 6 7 8
10 15		1 3 6 7 8 9
15		1 3 6 7 8 9 10
		1 3 6 7 8 9 10 15

For example, in the first line, the leftmost numbers in the first and the second half are 1 and 3, respectively. They are compared to each other, and since 1 is smaller, it is taken out and written in the output.

In this example, exactly seven pairwise comparisons were made in the merging phase. In general, if the length of the sequence is n, the number of comparisons when merging the two halves is $n-1$.

We want to count the total number $C(n)$ of comparisons that Merge Sort performs when sorting a sequence of n numbers. We will assume that n is a power of two so that any time the sequence is split in half in a recursive call the two halves are equal.

To sort n numbers (for $n > 1$), Merge Sort makes two recursive calls to sequences of length $n/2$ and performs exactly $n-1$ comparisons in the merging phase. This gives the recurrence

$$C(n) = 2C(n/2) + (n-1) \qquad \text{for } n > 1 \tag{3}$$

with the base case $C(1) = 0$.

Let's try to guess a solution for this recurrence by working backwards as usual:

$$\begin{aligned} C(n) &= 2C(n/2) + (n-1) \\ &= 2(2C(n/2^2) + (n/2-1)) + (n-1) = 2^2 C(n/2^2) + 2n - (2+1) \\ &= 2^2(2C(n/2^3) + n/2^2 - 1) + 2n - (2+1) = 2^3 C(n/2^3) + 3n - (2^2+2+1). \end{aligned}$$

We reach $C(1)$ on the right hand side after $\log n$ steps.[1] The expression we get on the right is

$$nC(1) + (\log n)n - (2^{\log n - 1} + \cdots + 2 + 1).$$

By our base case, $C(1) = 0$. By the formula for geometric sums, the last term equals $2^{\log n} - 1 = n - 1$. This suggests the guess

$$C(n) = n \log n - n + 1.$$

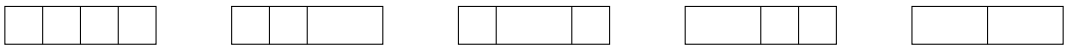
This formula indeed sets $C(1)$ to zero as desired. You can now verify on your own that this guess solves the recursion by a calculation.

Theorem 3. *The assignment $C(n) = n \log n - n + 1$ satisfies the equations (3).*

[1] In computer science, we use log for base two logarithms.

4 Homogeneous linear recurrences

You are given an unlimited supply of 1×1 and 2×1 tiles. In how many ways can you tile a hallway of dimension $n \times 1$ using the tiles? For example, here are all five possible tilings for $n = 4$:



Let $f(n)$ denote the number of desired tilings of an $n \times 1$ hallway. If $n = 0$ there is exactly one possible tiling — use no tiles — so $f(0) = 1$. If $n = 1$ there is also one tiling — use a single 1×1 tile — so $f(1) = 1$.

When $n \geq 2$, we distinguish two possible types of tilings. If the tiling starts with a 1×1 tile, then the remaining part of the corridor can be tiled in $f(n-1)$ ways. If the tiling starts with a 2×1 tile, then there are $f(n-2)$ possible tilings. Since these two possibilities cover each possibility exactly once, we obtain the recurrence

$$f(n) = f(n-1) + f(n-2) \qquad \text{for every } n \geq 2. \tag{4}$$

The numbers $f(0), f(1), f(2), \ldots$ are no other than the Fibonacci numbers. The formula (4) is a *homogeneous linear recurrence*. Such recurrences can be solved using the following guess-and-verify method.

At first, we forget about the "base cases" $f(0) = 0, f(1) = 1$ and focus on the equations (4). We look for solutions of the type $f(n) = x^n$ for some nonzero real number x. The reason we expect such solutions to work out is because for every $n \geq 2$, the equation $x^n = x^{n-1} + x^{n-2}$ is the same as

$$x^2 = x + 1$$

since all we did was scale down both sides by a factor of x^{n-2}. This is a quadratic equation in x and we can use the quadratic formula to calculate its solutions

$$x_1 = \frac{1+\sqrt{5}}{2} \qquad \text{and} \qquad x_2 = \frac{1-\sqrt{5}}{2}.$$

It is easy to check that both $f(n) = x_1^n$ and $f(n) = x_2^n$ satisfy all the equations (4). But what about the requirements $f(0) = 1$ and $f(1) = 1$? Neither solution satisfies these additional conditions.

Every *linear combination* $f(n) = sx_1^n + tx_2^n$ also solves the equations (4), so if we can find numbers s and t such that

$$1 = f(0) = sx_1^0 + tx_2^0 = s + t$$

and

$$1 = f(1) = sx_1^1 + tx_2^1 = s \cdot \frac{1+\sqrt{5}}{2} + t \cdot \frac{1-\sqrt{5}}{2}$$

then the assignment $f(n) = sx_1^n + tx_2^n$ must be the solution to our problem.

To find s and t, we need to solve two equations with two unknowns. These equations are simple enough that we can solve them by hand; no need for the computer. After subtracting the first equation scaled by $1/2$ from the second one we get

$$\frac{1}{2} = \frac{\sqrt{5}}{2}(s - t)$$

or $s - t = 1/\sqrt{5}$. Adding this to $s + t = 1$ we get $2s = 1 + 1/\sqrt{5}$, from where

$$s = \frac{1}{2} + \frac{1}{2\sqrt{5}} = \frac{1}{\sqrt{5}} \cdot \frac{1+\sqrt{5}}{2}$$

and

$$t = 1 - s = -\frac{1}{\sqrt{5}} \cdot \frac{1-\sqrt{5}}{2}$$

from where

$$f(n) = sx_1^n + tx_2^n = \frac{1}{\sqrt{5}} \cdot \left(\frac{1+\sqrt{5}}{2}\right)^{n+1} - \frac{1}{\sqrt{5}} \cdot \left(\frac{1-\sqrt{5}}{2}\right)^{n+1}. \tag{5}$$

You can now verify in the usual way that this solution is indeed correct.

Theorem 4. *The assignment (5) satisfies the equations (4) with initial conditions $f(0) = 1$, $f(1) = 1$.*

The formula (5) tells us how these numbers behave when n is large. Since $(1 + \sqrt{5})/2 \approx 1.618$ and $(1 - \sqrt{5})/2 \approx -0.618$, the term $((1 + \sqrt{5})/2)^n$ will grow when n becomes large and the term $((1 - \sqrt{5})/2)^n$ will become vanishingly small; eventually, it becomes smaller than any fixed constant. Therefore we can write

$$f(n) = \frac{1}{\sqrt{5}} \cdot \left(\frac{1 + \sqrt{5}}{2}\right)^{n+1} + o(1).$$

For an even rougher estimate, we can disregard all the constants in the first term to get

$$f(n) = \Theta\left(\left(\frac{1 + \sqrt{5}}{2}\right)^n\right).$$

In general, this method can be used to solve any homogeneous linear recurrence, that is a recurrence of the type

$$f(n) = a_1 f(n-1) + a_2 f(n-2) + \cdots + a_k f(n-k)$$

as long as k "base cases" are provided. If you can't figure out the procedure, please see Section 22.3.2 of the textbook.

5 Homogenization

The recurrence that counts the number of moves for the Towers of Hanoi

$$T(n) = 2T(n-1) + 1$$

is *not* homogeneous because of the extra $+1$ term on the right hand side. It can be turned into a homogeneous recurrence like this: If I add another $+1$ to both sides I can rewrite this equation as

$$T(n) + 1 = 2(T(n-1) + 1)$$

Now we can introduce the "change of variables" $T'(n) = T(n) + 1$ and get the homogeneous recurrence

$$T'(n) = 2T'(n-1)$$

which is then easy to solve: $T'(n) = 2^n T'(0) = 2^n$, so $T(n) = T'(n) - 1 = 2^n - 1$.

Labors of Hanoi We might expect that moving a small disk requires less work than moving a larger one. Let's assume that moving the smallest disk uses up one unit of work, the second smallest uses two units, and so on. What is the total amount of work $W(n)$ used for moving the whole n-disk stack? By similar logic, the recurrence for work is

$$W(n) = 2W(n-1) + n \tag{6}$$

with base case $W(1) = 1$. This recurrence is not homogeneous. We may try to homogenize it by adding some factors of n to both sides. How to we find these factors? We can attempt a general guess of the form $W(n) = W'(n) + a \cdot n$ for some (for now unknown) constant a. The recurrence then becomes

$$W(n) + a \cdot n = 2(W(n-1) + a \cdot (n-1)) + n.$$

In order to homogenize the recurrence, it better be the case that all terms involving a vanish, namely

$$a \cdot n = 2 \cdot a \cdot (n-1) + n \qquad \text{for all } n.$$

We can try to solve the equation for a by grouping terms:

$$(a+1)\cdot n - 2a = 0.$$

As this equation should hold for all n, it better be the case that $a+1=0$ and $2a=0$. This is clearly impossible. This attempt to homogenize (8) was not successful. We can try again with a more general guess of the form

$$W(n) = W'(n) + a\cdot n + b.$$

Plugging into (6) we get

$$W(n) + a\cdot n + b = 2(W(n-1) + a\cdot(n-1) + b) + n.$$

To homogenize the recurrence we would need

$$a\cdot n + b = 2\cdot(a\cdot(n-1)+b) + n \qquad \text{for all } n,$$

or

$$(a+1)\cdot n - 2a + b = 0 \qquad \text{for all } n.$$

There is now a solution, namely $a=-1$ and $b=-2$. Therefore, if we set

$$W(n) = W'(n) - n - 2 \tag{7}$$

then W' satisfies the recurrence $W'(n) = 2W'(n-1)$. This *homogenized recurrence* solves to $W'(n) = 2^{n-1}W'(1)$.

The base case for W' can be obtained by using (7) backwards: $W'(1) = W(1)+1+2 = 1+1+2 = 4$, so $W'(n) = 2^{n-1}\cdot 4 = 2^{n+1}$ and the original recurrence solves to

$$W(n) = 2^{n+1} - n - 2.$$

Homogenizing linear recurrences Let's try another nonhomogeneous recurrence:

$$f(n) = f(n-1) + f(n-2) + n \tag{8}$$

with initial conditions $f(0)=-2$, $f(1)=-3$. We may try to homogenize it by adding some factors of n to both sides. Based on the previous example, we may venture a general guess of the form

$$f(n) = g(n) + a\cdot n + b$$

with unknown constants a and b. Plugging into (8) we get

$$g(n) + a\cdot n + b = \big(g(n-1) + a\cdot(n-1) + b\big) + \big(g(n-2) + a\cdot(n-2) + b\big) + n.$$

To homogenize the recurrence we would need

$$a\cdot n + b = a\cdot(n-1) + b + a\cdot(n-2) + b + n \qquad \text{for all } n$$

or

$$(a+1)\cdot n - 3a + b = 0.$$

There is now a solution, namely $a=-1$ and $b=-3$. Therefore, if we set $f(n) = g(n) - n - 3$ then g satisfies the recurrence $g(n) = g(n-1) + g(n-2)$. Plugging in $n=0$ and $n=1$ the initial conditions for g are $g(0)=1$ and $g(1)=1$. This is exactly the Fibonacci sequence, so

$$g(n) = \frac{1}{\sqrt{5}}\cdot\Big(\frac{1+\sqrt{5}}{2}\Big)^{n+1} - \frac{1}{\sqrt{5}}\cdot\Big(\frac{1-\sqrt{5}}{2}\Big)^{n+1}$$

and therefore

$$f(n) = g(n) - n - 3 = \frac{1}{\sqrt{5}}\cdot\Big(\frac{1+\sqrt{5}}{2}\Big)^{n+1} - \frac{1}{\sqrt{5}}\cdot\Big(\frac{1-\sqrt{5}}{2}\Big)^{n+1} - n - 3.$$

6 Analysis of Euclid's algorithm

In Lecture 4 we said that one advantage of Euclid's algorithm over the "schoolbook gcd" is that the former is a lot faster when the inputs are big. But how fast is Euclid's algorithm?

When we measure the speed of algorithms we have to first agree on what counts for a single step. In the case of Euclid's algorithm it is sensible to count the number of transitions $(a, b) \to (b, a \bmod b)$ of the underlying state machine (assuming $a \geq b$), namely the number of divisions with remainder that the algorithm performs.

This number will clearly depend on the inputs n and d whose gcd is being computed. We expect that as the inputs become larger, the number of steps should grow. This intuition is not completely accurate. For example, when the inputs are 27 and 13,

$$E(27, 13) = E(13, 1) = E(1, 0) = 1$$

and Euclid's algorithm terminates in two steps. When the inputs are 21 and 13, however, the transitions are

$$E(21, 13) = E(13, 8) = E(8, 5) = E(5, 3) = E(3, 2) = E(2, 1) = E(1, 0) = 1 \tag{9}$$

and the algorithm takes six steps to terminate.

These examples suggests that it may be tricky to derive an exact formula for the complexity of Euclid's algorithm. Fortunately it turns out that it is not too difficult to obtain a good *upper bound* on the number of steps. Let (a_{t+1}, a_t) denote the state of Euclid's algorithm with t steps remaining to termination. For example, in execution (9) the initial state is $(a_7 = 22, a_6 = 13)$ and the sequence $a_7, a_6, \ldots, a_0$ is $a_7 = 22$, $a_6 = 13$, $a_5 = 8$, $a_4 = 5$, $a_3 = 3$, $a_2 = 2$, $a_1 = 1$, and $a_0 = 0$. As a_{t-1} is always the remainder of dividing a_{t+1} by a_t, it must be that

$$a_{t-1} \leq a_{t+1} - a_t \qquad \text{for } t = 1 \text{ up to } n - 1.$$

Reversing this inequality it follows that

$$a_{t+1} \geq a_t + a_{t-1} \qquad \text{for } t = 1 \text{ up to } n - 1.$$

This looks a lot like the definition of the Fibonacci sequence, except that the equality is replaced with an inequality. Although the numbers a_1, a_2 and so on are not determined by these inequality, we can say that they grow at least as fast as the Fibonacci numbers.

Lemma 5. *$a_t \geq f(t-1)$ for all t from 0 up to n (where we set $f(-1) = 0$).*

Proof. We apply strong induction on t. The final state of Euclid's algorithm is (a_1, a_0) with $a_1 > 0$ and $a_0 = 0$. Therefore $a_0 \geq f(-1)$ and $a_1 \geq f(0)$ proving the base cases. For the inductive step we assume that $a_0 \geq f(-1)$, $a_1 \geq f(0)$, $a_2 \geq f(2)$, up to $a_t \geq f(t)$. Then $a_{t+1} \geq a_t + a_{t-1} \geq f(t) + f(t-1) = f(t+1)$ as desired. □

Now suppose Euclid's algorithm runs for n steps. By this Lemma the smaller of the two inputs $a = a_n$ must be at least as large as the n-th Fibonacci number, so

$$a_n \geq f(n) = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{1}{\sqrt{5}} \cdot \left(\frac{1-\sqrt{5}}{2}\right)^n.$$

therefore the number of steps n can be at most

$$n \leq \log_\phi\left(\sqrt{5} \cdot a_n + \left(\frac{1-\sqrt{5}}{2}\right)^n\right) \leq \log_\phi(\sqrt{5} \cdot a_n + 1),$$

where $\phi = (1 + \sqrt{5})/2 \approx 1.618$. For example, if the larger input a_n is 2^{300} then $\log_\phi(\sqrt{5}a_n + 1) \leq 434$. Even with such a huge input Euclid's algorithm will terminate quite fast.

If we do not care much about exact constants, we can write

$$\log_\phi(\sqrt{5} \cdot a_n + 1) = \log_\phi(O(a_n)) = \log_\phi(a_n) + O(1).$$

Up to some additive constant, the complexity of Euclid's algorithm is at most $\log_\phi$(smaller of the two inputs).

7 The Master Theorem

Divide-and-conquer is a paradigm for solving problems that works like this: We split a problem of a large size into a fixed number of subproblems of a fraction of its size and combine their solutions in some manner to obtain a solution to the bigger problem. Merge Sort is an example of a divide-and-conquer algorithm.

In general, if a problem of size n is split into a subproblems of size n/b, the recurrence that arises in its analysis has the form

$$T(n) = aT(n/b) + g(n). \tag{10}$$

Here, $g(n)$ describes the "complexity" of combining the solutions to the subproblems. For example, in Merge-Sort we split a problem of size n into two problems of size $n/2$ and combined their solutions using $n-1$ comparisons to obtain the recurrence $C(n) = 2C(n/2) + n - 1$, thus $a = 2$, $b = 2$, and $g(n) = n - 1$.

There is a general rule that tells us the *asymptotic behavior* of the function $T(n)$ that satisfies the recurrence (10) for common choices of the function g:

The Master Theorem. Let $c = \log_b a$. Then

$$T(n) \text{ is } \begin{cases} \Theta(n^c), & \text{if } g(n) \text{ is } O(n^{c-\varepsilon}) \text{ for some } \varepsilon > 0, \\ \Theta(n \log n), & \text{if } g(n) \text{ is } \Theta(n^c), \\ \Theta(g(n)), & \text{if } n^{c+\varepsilon} \text{ is } O(g(n)) \text{ for some } \varepsilon > 0. \end{cases}$$

In the Merge Sort recurrence, $c = \log_b a = \log_2 2 = 1$ so $g(n)$ is $\Theta(n)$ and the Master Theorem tells us that the recurrence solves to $C(n) = \Theta(n \log n)$ as we already know.

Here is another example. The following recurrence arises in the analysis of the running time of a an algorithm for multiplying matrices that was invented by Volker Strassen:

$$M(n) = 7M(n/2) + \Theta(n^2).$$

In this case, we do not even know what the function g is precisely, but only that it grows asymptotically as n^2. Here, $c = \log_b a = \log_2 7 \approx 2.807$. As n^2 is $O(n^{2.806})$, the first case applies and we get that $M(n)$ is $\Theta(n^{\log_2 7})$.

References

This lecture is based on Chapter 21 of the text *Mathematics for Computer Science* by E. Lehman, T. Leighton, and A. Meyer.