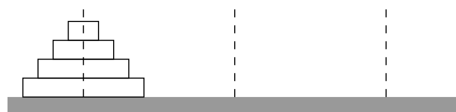
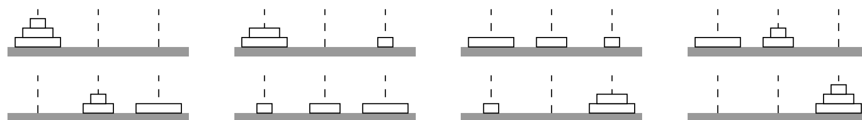
Recurrences are formulas that describe the value of a function of positive integers at an input in terms of the value of the same function at smaller inputs. We will see examples of problems in which recurrences come about and show how to solve them.

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 $T(n)$ 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

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

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 appear to 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

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

Let's do another example. Recall the Beneš network B_n from Lecture 6. How many vertices does the digraph B_n have? Recall that B_n was constructed recursively as follows. B_1 is a network with two sources and two sinks (and a directed edge from each source to sink). The network B_{n+1} was constructed by taking two disjoint copies of B_n, adding 2^{n+1} new sources, 2^{n+1} new sinks — 2^{n+2} new vertices in total — and connecting them up in some way. So the number of vertices $V(n)$ of B_n is given by the recurrence

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

Let us try to solve this recurrence by working backwards:

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

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

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

We can now verify that our guess is correct:

Theorem 2. *For all $n \geq 1$, the digraph B_n has $V(n) = n \cdot 2^{n+1}$ vertices.*

Proof. We prove the theorem by induction on n. The digraph B_1 has $4 = 1 \cdot 2^2$ vertices as desired. Now assume B_n has $n2^{n+1}$ vertices. By recurrence (1), B_{n+1} then has

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

vertices as desired. □

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:

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

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

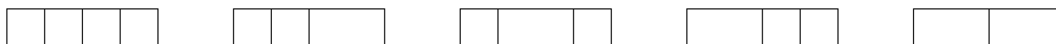
$$C(n) = n \log n - n + 1.$$

This formula indeed sets $T(1)$ 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).*

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 $n = 4$:



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

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

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

This is an example of a *homogeneous linear recurrence*. Such recurrences can be solved using the following guess verify method.

Initially, we forget about the "base cases" $f(0) = 1, f(1) = 1$ 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 o 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 that $f(0) = 1$ and $f(1) = 1$? Neither solution satisfies these additional conditions.

Now we notice that 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

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

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

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

The number $f(n)$ is called the *n-th Fibonacci number*. 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

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

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

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

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 10.3.3 of the textbook.

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.

5 Divide-and-conquer recurrences

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_1 subproblems of size $b_1 n$, a_2 subproblems of size $b_2 n$, up to a_k subproblems of size $b_k n$ (where $b_1, \dots, b_k$ are constants between 0 and 1) the recurrences that arise in its analysis have the form

$$T(n) = a_1 T(b_1 n) + \cdots + a_k T(b_k n) + g(n). \tag{6}$$

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 $C(n) = 2C(n/2) + n - 1$.

There is a general formula that tells us the *asymptotic behavior* of the function $T(n)$ that satisfies the recurrence (6), assuming that the function $g(n)$ is "sufficiently nice." (We'll explain precisely what this means shortly.) The asymptotic behaviour of T is given by the formula

$$T(n) = \Theta\left(n^p \left(1 + \int_1^n \frac{g(u)}{u^{p+1}}\, du\right)\right)$$

where p is the unique constant such that $a_1 b_1^p + \cdots + a_k b_k^p = 1$.

For example, in the Merge Sort recurrence, $k = 1$, $a_1 = 2$ and $b_1 = 1/2$ so p is the number that solves the equation $2 \cdot (1/2)^p = 1$, namely $p = 1$. Then

$$\int_1^n \frac{u-1}{u^{p+1}} = \int_1^n \frac{u-1}{u^2} = \ln u + \frac{1}{u}\Big|_1^n = \ln n - 1 + \frac{1}{n}$$

The dominant term here is $\ln n$, so $C(n) = \Theta(n \log n)$, as we already know.

Let's do another example. The following recurrence arises in the analysis of the running time of a certain algorithm for multiplying matrices:

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

In this case, we do not even know what the function g is precisely, but only that it grows asymptotically as n^2. The value p needs to satisfy $7 \cdot (1/2)^p = 1$, so $p = \log_2 7$. Let us now calculate the relevant integral, first assuming that $g(x)$ is *exactly* equal to x^2:

$$\int_1^n \frac{u^2}{u^{p+1}}\, du = \int_1^n u^{1-\log_2 7}\, du = O(1)$$

because the exponent is $1 - \log_2 7 \approx 1.807$, which is greater than one, and so the integral converges. (You can come to the same conclusion by calculating the integral.)

The actual integral $\int_1^n g(u)/u^{p+1}\,du$ may not have the exact same value, but since $g(u)$ is within a constant factor of u^2 (when u is sufficiently large), so will be the two integrals and we can conclude that $\int_1^n g(u)/u^{p+1}\,du = O(1)$. Therefore

$$M(n) = \Theta\left(n^p\left(1 + \int_1^n \frac{g(u)}{u^{p+1}}\,du\right)\right) = \Theta(n^p) = \Theta(n^{\log_2 7}).$$

Here is a precise statement of the theorem that allows us to obtain the asymptotic behaviour of solutions to a large number of divide-and-conquer recurrences.

Theorem 5. *Assume the function $T(x)$ is nonnegative and bounded when $0 \le x \le x_0$ and satisfies the recurrence*

$$T(x) = a_1T(b_1x) + \cdots + a_kT(b_kx) + g(x)$$

for $x > 0$ for some positive constants $a_1, \ldots, a_k$ and constants $b_1, \ldots, b_k$ between zero and one.

If g is non-negative and $|dg(x)/dx|$ is $O(x^C)$ for some constant C, then

$$T(n) = \Theta\left(n^p\left(1 + \int_1^n \frac{g(u)}{u^{p+1}}\,du\right)\right)$$

where p is the unique constant such that $a_1b_1^p + \cdots + a_kb_k^p = 1$.

In the Merge Sort example, $g(x) = x - 1$ so $|dg(x)/dx| = 1$ and our use of Theorem 5 is legitimate. In the other example, we can say $M(n) = \Theta(n^{\log_2 7})$ assuming that the function represented by $\Theta(n^2)$ in the definition of M has the absolute value of its derivative bounded by a polynomial.

The textbook states a more general version of the theorem accounting for the possibility that the subproblem sizes might not be exactly $b_1n, \ldots, b_kn$ but perhaps deviate a bit from these values. Instead of showing the general version, let me just say that when this deviation is sufficiently small (e.g. a constant) then the theorem continues to hold.

References

This lecture is based on Chapter 10 of the text *Mathematics for Computer Science* by E. Lehman, T. Leighton, and A. Meyer.

Theorem 5 was discovered by Mohammad Akra and Louay Bazzi in 1996. You can read more about it and find its proof in these notes by Tom Leighton.