Recurrences

A side note about induction

We have seen many examples with induction used to prove summation formulas. But we have to make it clear: *Induction does not have to be applied to summations* or any arithmetic expressions. It is a much more general approach:

$$\frac{\begin{array}{l} P(0) \\ P(n) \rightarrow P(n+1) \text{ for all } n \geq 0 \end{array}}{P(k) \text{ for all } k \geq 0}$$

For example, the problem, where we were tiling checkerboards. There were no summation, but it was a good example of induction.

Recurrence

Recall that we used induction to prove statements like

$$\sum_{k=0}^{n} k = 0 + 1 + 2 + 3 + \ldots + n = \frac{n(n+1)}{2}$$

In problems like this, we used a common pattern:

$$\sum_{k=0}^{0} k = 0$$

$$\sum_{k=0}^{n} k = \left(\sum_{k=0}^{n-1} k\right) + n, \quad \text{when } n > 0$$

That is, we can express the sum of natural numbers recursively in terms of a smaller sum.

Recurrence

Let $S(n)$ be the sum of all natural numbers not greater than n:

$$S(n) = \sum_{k=0}^{n} k,$$

It can be convenient to redefine the sum $S(n)$ as a *recurrence*:

$$\begin{aligned} S(0) &= 0 \\ S(n) &= S(n-1) + n \qquad (\forall n > 0) \end{aligned}$$

This is just another way to express the same function S.

Recurrence

Exponentiation:

$$E(a,n) = a^n$$

Recursively:

$$\begin{aligned} E(a,0) &= 1 \\ E(a,n) &= E(a,n-1)\cdot a \qquad (\forall n > 0) \end{aligned}$$

Recurrence

Factorial:

$$n! = 1 \cdot 2 \cdot \ldots \cdot n$$

Recursively:

$$\begin{aligned} 0! &= 1 \\ n! &= (n-1)! \cdot n \qquad (\forall n > 0) \end{aligned}$$

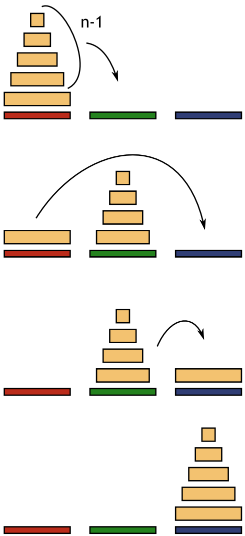
The towers of Hanoi

`http://www.mathsisfun.com/games/towerofhanoi.html`

The towers of Hanoi

Our recursive algorithm to move a tower of height n from #1 to #3:

1. Move an $(n-1)$-tower from #1 to #2.
2. Move an 1-tower from #1 to #3.
3. Move an $(n-1)$-tower from #2 to #3.



The towers of Hanoi

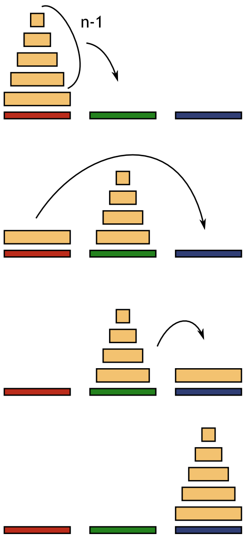
Our recursive algorithm to move a tower of height n from #1 to #3:

1. Move an $(n-1)$-tower from #1 to #2.

2. Move an 1-tower from #1 to #3.

3. Move an $(n-1)$-tower from #2 to #3.

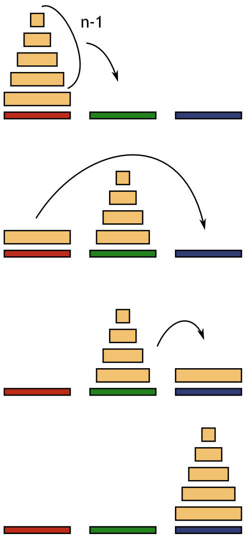
There is a way to find a recurrent formula for T_n, the total number of steps to move the tower from the peg 1 to the peg 3.



The towers of Hanoi

T_n, the time to move a tower of height n:

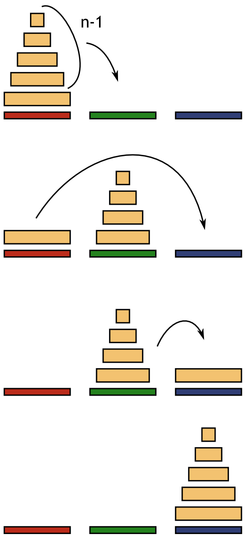
$$T_1 = 1$$
$$T_n = T_{n-1} + 1 + T_{n-1} \qquad (\forall n > 1)$$



The towers of Hanoi

T_n, the time to move a tower of height n:

$$T_1 = 1$$
$$T_n = T_{n-1} + 1 + T_{n-1} \qquad (\forall n > 1)$$



There is a proof by induction that this time is optimal for any algorithm.

The towers of Hanoi

$$T_1 = 1$$
$$T_n = 2T_{n-1} + 1 \qquad (\forall n > 1)$$

Our goal is to find a closed form expression for T_n as a function of n, without any recurrence.

The towers of Hanoi

Before we get a closed form formula for T_n, what are the numbers?

$$\begin{aligned} T_1 &= 1 \\ T_n &= 2T_{n-1} + 1 \qquad (\forall n > 1) \end{aligned}$$

We can compute a list like this:

$$\begin{aligned} T_1 &= 1 \\ T_2 &= 3 \\ T_3 &= 7 \\ T_4 &= 15 \\ T_5 &= 31 \\ T_6 &= 63 \\ &\dots \end{aligned}$$

The towers of Hanoi

$$
\begin{aligned}
T_1 &= 1 \\
T_n &= 2T_{n-1} + 1 \qquad (\forall n > 1)
\end{aligned}
$$

$$
\begin{aligned}
T_1 &= 1 \\
T_2 &= 3 \\
T_3 &= 7 \\
T_4 &= 15 \\
T_5 &= 31 \\
T_6 &= 63 \\
&\dots
\end{aligned}
$$

Guess and verify method... Let's try $T_n = 2^n - 1$?

The towers of Hanoi

$$
\begin{aligned}
T_1 &= 1 \\
T_n &= 2T_{n-1} + 1 \qquad (\forall n > 1)
\end{aligned}
$$

Guess and verify method... Let's try $T_n = 2^n - 1$? We can show by induction that this formula is correct.

The base case, $n = 1$:

$$T_1 = 2^1 - 1 = 1.$$

Ok, the base case is true.

The towers of Hanoi

$$T_1 = 1$$
$$T_n = 2T_{n-1} + 1 \qquad (\forall n > 1)$$

We want to prove the closed form formula $T_n = 2^n - 1$.

The inductive step, $n > 1$:
Assume that $T_n = 2^n - 1$, and show that then $T_{n+1} = 2^{n+1} - 1$.

Proof. From the recurrence:

$$T_{n+1} = 2T_n + 1$$

By the inductive hypothesis:

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

The towers of Hanoi

Why is it useful to know that the recurrence

$$T_1 = 1$$
$$T_n = 2T_{n-1} + 1 \qquad (\forall n > 1)$$

is equivalent to the closed form formula $T_n = 2^n - 1$?

The 7-disk puzzle will require $T_7 = 2^7 - 1 = 127$ moves to complete.

And the 100-disk puzzle will require

$T_{100} = 2^{100} - 1 = 1267650600228229401496703205375$ moves.

Merge sort

function Merge:
Given two sorted lists, combine them into a single sorted list:

$$[1, 2, 4, 5] + [3, 4, 5, 6] \mapsto [1, 2, 3, 4, 4, 5, 5, 6]$$

function Sort:
Given a list: if it cantains a single element, return it. Otherwise, split it in two halves sort them separately and merge the results:

$$\mathbf{S}[5] \mapsto [5]$$

$$\mathbf{S}[6, 7, 1, 8, 9, 7, 4, 3] \mapsto \mathbf{S}[6, 7, 1, 8] + \mathbf{S}[9, 7, 4, 3]$$

Merge sort

$$\mathbf{S}[1,8,3,6,5,4,7,2] \mapsto$$

Merge sort

$$\mathbf{S}[1,8,3,6,5,4,7,2] \mapsto$$

$$\mathbf{S}[1,8,3,6] + \mathbf{S}[5,4,7,2] \mapsto$$

Merge sort

$$\mathbf{S}[1,8,3,6,5,4,7,2] \mapsto$$

$$\mathbf{S}[1,8,3,6] + \mathbf{S}[5,4,7,2] \mapsto$$

$$\big(\mathbf{S}[1,8] + \mathbf{S}[3,6]\big) + \big(\mathbf{S}[5,4] + \mathbf{S}[7,2]\big) \mapsto$$

Merge sort

$$\mathbf{S}[1,8,3,6,5,4,7,2] \mapsto$$

$$\mathbf{S}[1,8,3,6] + \mathbf{S}[5,4,7,2] \mapsto$$

$$\big(\mathbf{S}[1,8] + \mathbf{S}[3,6]\big) + \big(\mathbf{S}[5,4] + \mathbf{S}[7,2]\big) \mapsto$$

$$\big((\mathbf{S}[1] + \mathbf{S}[8]) + (\mathbf{S}[3] + \mathbf{S}[6])\big) + \big((\mathbf{S}[5] + \mathbf{S}[4]) + (\mathbf{S}[7] + \mathbf{S}[2])\big) \mapsto$$

Merge sort

$$\mathbf{S}[1,8,3,6,5,4,7,2] \mapsto$$
$$\mathbf{S}[1,8,3,6] + \mathbf{S}[5,4,7,2] \mapsto$$
$$\big(\mathbf{S}[1,8] + \mathbf{S}[3,6]\big) + \big(\mathbf{S}[5,4] + \mathbf{S}[7,2]\big) \mapsto$$
$$\big((\mathbf{S}[1] + \mathbf{S}[8]) + (\mathbf{S}[3] + \mathbf{S}[6])\big) + \big((\mathbf{S}[5] + \mathbf{S}[4]) + (\mathbf{S}[7] + \mathbf{S}[2])\big) \mapsto$$
$$\big(([1] + [8]) + ([3] + [6])\big) + \big(([5] + [4]) + ([7] + [2])\big) \mapsto$$

Merge sort

$$\mathbf{S}[1,8,3,6,5,4,7,2] \mapsto$$

$$\mathbf{S}[1,8,3,6] + \mathbf{S}[5,4,7,2] \mapsto$$

$$\big(\mathbf{S}[1,8] + \mathbf{S}[3,6]\big) + \big(\mathbf{S}[5,4] + \mathbf{S}[7,2]\big) \mapsto$$

$$\big((\mathbf{S}[1] + \mathbf{S}[8]) + (\mathbf{S}[3] + \mathbf{S}[6])\big) + \big((\mathbf{S}[5] + \mathbf{S}[4]) + (\mathbf{S}[7] + \mathbf{S}[2])\big) \mapsto$$

$$\big(([1] + [8]) + ([3] + [6])\big) + \big(([5] + [4]) + ([7] + [2])\big) \mapsto$$

$$\big([1,8] + [3,6]\big) + \big([4,5] + [2,7]\big) \mapsto$$

Merge sort

$$\mathbf{S}[1,8,3,6,5,4,7,2] \mapsto$$

$$\mathbf{S}[1,8,3,6] + \mathbf{S}[5,4,7,2] \mapsto$$

$$\big(\mathbf{S}[1,8] + \mathbf{S}[3,6]\big) + \big(\mathbf{S}[5,4] + \mathbf{S}[7,2]\big) \mapsto$$

$$\big((\mathbf{S}[1] + \mathbf{S}[8]) + (\mathbf{S}[3] + \mathbf{S}[6])\big) + \big((\mathbf{S}[5] + \mathbf{S}[4]) + (\mathbf{S}[7] + \mathbf{S}[2])\big) \mapsto$$

$$\big(([1] + [8]) + ([3] + [6])\big) + \big(([5] + [4]) + ([7] + [2])\big) \mapsto$$

$$\big([1,8] + [3,6]\big) + \big([4,5] + [2,7]\big) \mapsto$$

$$[1,3,6,8] + [2,4,5,7] \mapsto$$

Merge sort

$$\mathbf{S}[1,8,3,6,5,4,7,2] \mapsto$$

$$\mathbf{S}[1,8,3,6] + \mathbf{S}[5,4,7,2] \mapsto$$

$$\big(\mathbf{S}[1,8] + \mathbf{S}[3,6]\big) + \big(\mathbf{S}[5,4] + \mathbf{S}[7,2]\big) \mapsto$$

$$\big((\mathbf{S}[1] + \mathbf{S}[8]) + (\mathbf{S}[3] + \mathbf{S}[6])\big) + \big((\mathbf{S}[5] + \mathbf{S}[4]) + (\mathbf{S}[7] + \mathbf{S}[2])\big) \mapsto$$

$$\big(([1] + [8]) + ([3] + [6])\big) + \big(([5] + [4]) + ([7] + [2])\big) \mapsto$$

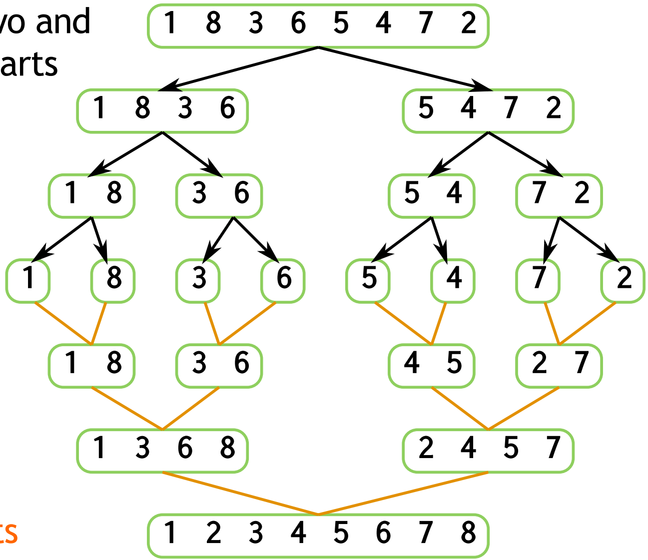
$$\big([1,8] + [3,6]\big) + \big([4,5] + [2,7]\big) \mapsto$$

$$[1,3,6,8] + [2,4,5,7] \mapsto$$

$$[1,2,3,4,5,6,7,8]$$

Merge sort

Split in two and
sort the parts



Merge
sorted lists

Merge sort

How much time does it take to sort a list of n elements?

To estimate the time complexity, we are going to *count the number of comparisons* between the elements.

We assume that the size of the given list is a power of 2. It makes the analysis easier, but does not affect the result.

Merge sort

(a) To merge two lists of size $n/2$, we need to do at most $n-1$ comparisons.

(b) To sort a list, we have to split it in two, sort both halves, and merge them.

Therefore,

$$
\begin{aligned}
&T(1) = 0 \\
&T(n) = 2T(n/2) + n - 1 \qquad (\forall n > 1)
\end{aligned}
$$

Merge sort

Given

$$T(n) = 2T(n/2) + n - 1 \qquad (\forall n > 1)$$

Since $n = 2^k$,

$$\begin{aligned}
T(n) = T(2^k) &= 2T(2^{k-1}) + (2^k - 1) \\
&= 2\left(2T(2^{k-2}) + 2^{k-1} - 1\right) + (2^k - 1) \\
&= 2^2 T(2^{k-2}) + (2^k - 2) + (2^k - 1) \\
&= 2^2\left(2T(2^{k-3}) + 2^{k-2} - 1\right) + (2^k - 2) + (2^k - 1) \\
&= 2^3 T(2^{k-3}) + (2^k - 4) + (2^k - 2) + (2^k - 1) \\
&= 2^3\left(2T(2^{k-4}) + 2^{k-3} - 1\right) + (2^k - 4) + (2^k - 2) + (2^k - 1) \\
&= 2^4 T(2^{k-4}) + (2^k - 8) + (2^k - 4) + (2^k - 2) + (2^k - 1) \\
&= \ldots = 2^k \underbrace{T(2^{k-k})}_{T(1)=0} + \sum_{i=0}^{k-1}(2^k - 2^i) = \sum_{i=0}^{k-1}(2^k - 2^i).
\end{aligned}$$

Merge sort

$$T(n) = T(2^k) = \sum_{i=0}^{k-1}(2^k - 2^i) = \sum_{i=0}^{k-1}(n - 2^i) = n \cdot k - \sum_{i=0}^{k-1} 2^i.$$

The sum of the geometric progression is

$$\sum_{i=0}^{k-1} 2^i = \frac{2^k - 1}{2 - 1} = 2^k - 1 = n - 1.$$

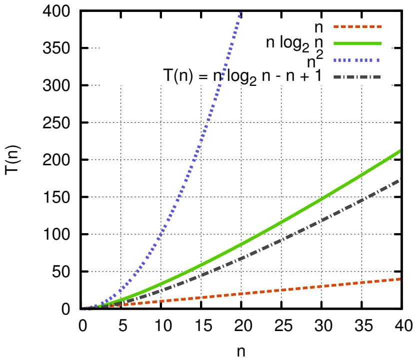
Thus $T(n) = n \cdot k - n + 1$. And since $n = 2^k$, $k = \log_2 n$, so

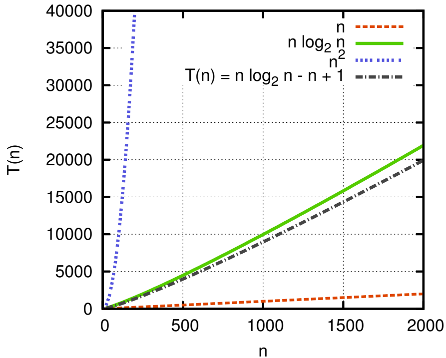
$$T(n) = n \log_2 n - n + 1.$$

Merge sort

To sort a list of length n, takes time (the number of comparisons)

$$T(n) = n\log_2 n - n + 1 \approx n\log_2 n.$$





Merge sort

In merge sort, we had a recurrence:

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

In general, if the time complexity of an algorithm is expressed by a recurrence:

$$T(n) = a \cdot T(n/b) + f(n)$$

To solve such recurrences, there is a so called *Master theorem*:
`https://en.wikipedia.org/wiki/Master_theorem`

It covers different forms of the function f, as well as difference values of the constants a and b.

Time complexity, big-O

Let's say that we've got this function as an estimation of the time complexity of an algorithm:

$$T(n) = 6n\log_2 n + 100n + \log_2 n + 50$$

Informally:

(a) If $T(n)$ is a sum, we take the fastest growing term only.

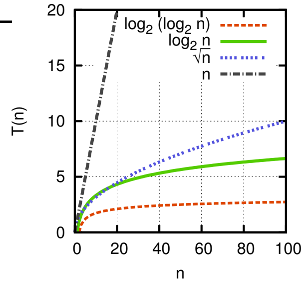
(b) We don't really care about constant factors.

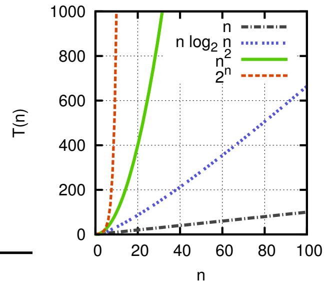
$$T(n) = O(n\log_2 n)$$

Time complexity, big-O

Some common time complexities, from the slowest to the fastest:

Running time	Name
$O(1)$	constant
$O(\log(\log n))$	log-logarithmic
$O(\log n)$	logarithmic
$O(\sqrt{n})$	square root (sub-linear)
$O(n)$	linear
$O(n \log n)$	n-log-n
$O(n^2)$	quadratic
$O(2^n)$	exponential
$O(n!)$	factorial
$O(2^{(2^n)})$	double exponential





Time complexity, big-O

Faster than linear algorithms, for example $O(\log n)$, cannot go through the whole input. They are cherry-picking in some sense, knowing where to search for the answer. Usually the input is structured in some way.

Example: *Binary search in a sorted array.*

Linear time algorithms, $O(n)$, usually have to read the whole input.

Example: *Search for the largest element in an unsorted array.*

Time complexity, big-O

Algorithms *slower than linear*, $O(n \log n)$ or $O(n^2)$, $O(n^7)$. Not only read the whole input, but also perform some extra work, but in a reasonably efficient way.

Example: *Sorting algorithms.*

Algorithms *much slower than linear*, exponential, for example, $O(2^n)$, are doing some non-trivial work.

Example: *Satisfiability of a statement in propositional logic.*

Time complexity, big-O

Formally:
We say that

$$T(n) = O(f(n))$$

if there are constants C and k such that

$$|T(n)| \leq C|f(n)| \qquad \text{for all } n > k$$

This definition says that after $n > k$, all slowly-growing terms don't really matter, and $T(n)$ behaves similarly to $f(n)$. To be more exact, $T(n)$ never exceeds $C \cdot f(n)$ when n is large enough.

$$T(n) = 6n \log_2 n + 100n + \log_2 n + 50$$

$$T(n) = O(n \log_2 n)$$