Last time...

Structural Induction.

↳ Let C be generated from B by F

if $\bullet\ \forall b \in B.\ P(b)$

$\bullet\ \forall f \in F,\ \forall a_1, \dots, a_m \in C.\ (P(a_1) \wedge \dots \wedge P(a_m) \Rightarrow P(f(a_1, \dots, a_m)))$

then $\forall x \in C.\ (P(x))$

Well-Ordering Principle

↳ $\forall S \subseteq \mathbb{N}$, if S is non-empty, S has a minimal element.

Annoucements

- See the recent Ed Post regarding the midterm
- Sign up for check in 2 by tonight. Reminder - you must do check ins in groups of 2 or 3, and all group members must be present unless an exception is requested.
- Tutorial change (more info at end)

CSC 236 Lecture 5: Recurrences 1

Harry Sha

June 7, 2023

Today

Recurrences

Asymptotics Review

Dominoes

Binary Search

Recurrences

A recursive function is one that depends on itself. Here are some examples.

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

Recursive ambiguity

There can be many functions that satisfy a single recurrence relation, for example
$F(n) = n + 5$ and $F(n) = n + 8$ both satisfy

$$F(n) = F(n-1) + 1$$

Thus, to specify a recursive function completely, we need to give it a (some) base case(s). I.e.

$$F(n) = \begin{cases} 5 & n = 0 \\ F(n-1) + 1 & n > 0 \end{cases}$$

Specifies the function $F(n) = n + 5$.

Why do we care about recurrences?

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

- The runtime of recursive programs can be expressed as recursive functions.
- Expressing something recursively is often an easier than expressing something explicitly.

The problem with recurrences

Here's the bad news. It's hard to answer questions like the following.

If

$$F(n) = \begin{cases} 2 & n = 0 \\ 7 & n = 1 \\ 2F(n-2) + F(n-1) + 12 & n > 1, \end{cases}$$

what is $F(100)$?

$n = \quad 0 \quad 1 \quad 2$

$F(n) \quad 2 \quad 7 \quad 23$

$4 + 7 + 12$

The problem with recurrences

Here's the bad news. It's hard to answer questions like the following.

If

$$F(n) = \begin{cases} 2 & n = 0 \\ 7 & n = 1 \\ 2F(n-2) + F(n-1) + 12 & n > 1, \end{cases}$$

what is $F(100)$?

We could do it - it would just take a while. Also, it is not immediately obvious what the asymptotics are. As computer scientists, we care about asymptotics. A lot.

Recursive to Explicit

Thus, once we have modelled something as a recurrence, it's still useful to convert that to an explicit definition of the same function.

Actually, since we're computer scientists, what we really care about is the asymptotics - we usually don't need a fully explicit expression.

Recursive to Explicit Examples

What is the explicit formula for

$$F(n) = \begin{cases} 1 & n = 0 \\ 2F(n-1) & n \geq 1 \end{cases}$$

n	0	1
	1	2

Recursive to Explicit Examples

What is the explicit formula for

$$F(n) = \begin{cases} 1 & n = 0 \\ 2F(n-1) & n \geq 1 \end{cases}$$

$F(n) = 2^n$. How can we prove it?

$P(n) = \quad F(n) = 2^n$

$P(0)$: $2^0 = 1 = F(0)$.

Inductive step. Let $k \in \mathbb{N}$, suppose $P(k)$ is. $F(k) = 2^k$.

$F(k+1) = 2 \cdot F(k) = 2 \cdot 2^k = 2^{k+1}$.

Recursive to Explicit Examples

What is the explicit formula for

$$F(n) = \begin{cases} 1 & n = 0 \\ 2F(n-1) & n \geq 1 \end{cases}$$

$F(n) = 2^n$. How can we prove it? By induction!

Recursive to Explicit Examples

$$F(n) = \begin{cases} 1 & n = 0 \\ 2F(n-1) & n \geq 1 \end{cases}$$

Recursive to Explicit Examples

What is the explicit formula for

$$F(n) = \begin{cases} 4 & n = 0 \\ 3 + F(n-1) & n \geq 1 \end{cases}$$

Claim: $3n+4$. ✓

n	0	1	2	?
F	4	7	10	

Recursive to Explicit Examples

What is the explicit formula for

$$F(n) = \begin{cases} 4 & n = 0 \\ 3 + F(n-1) & n \geq 1 \end{cases}$$

$F(n) = 4 + 3n$

Recursive to Explicit Examples

What is the explicit formula for

$$F(n) = \begin{cases} 1 & n = 0 \\ -F(n-1) & n \geq 1 \end{cases}$$

$(-1)^n$

Recursive to Explicit Examples

What is the explicit formula for

$$F(n) = \begin{cases} 1 & n = 0 \\ -F(n-1) & n \geq 1 \end{cases}$$

$F(n) = (-1)^n$

Recursive to Explicit Examples

What is the explicit formula for

$$F(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ 2F(n-1) - F(n-2) + 2 & n \geq 2 \end{cases}$$

Recursive to Explicit Examples

What is the explicit formula for

$$F(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ 2F(n-1) - F(n-2) + 2 & n \geq 2 \end{cases}$$

Claim. $F(n) = n^2$.

Proof

$$F(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ 2F(n-1) - \underline{F(n-2)} + 2 & n \geq 2 \end{cases}$$

Base case: $F(0) = 0 = 0^2$

$F(1) = 1 = 1^2$ ✓.

Inductive step: let $k \geq 1$, and assume $F(i) = i^2$ $\forall i \in \mathbb{N}$, $i \leq k$. WTS: $(k+1)^2 = F(k+1)$.

$$F(k+1) = 2F(k) - F(k-1) + 2.$$

$$= 2k^2 - (k-1)^2 + 2$$

$$= 2k^2 - (k^2 - 2k + 1) + 2$$

$$= k^2 + 2k + 1 = (k+1)^2$$

The functions we care about

Let's always imagine the function in question is the runtime of some algorithm. I.e., it maps the size of the input to the running time of the algorithm. Thus, we assume it has the following properties.

- domain $\mathbb{N}$.
- codomain $\mathbb{R}_{>0}$. An algorithm can't take negative time
- non-decreasing. An algorithm shouldn't get faster for larger inputs.

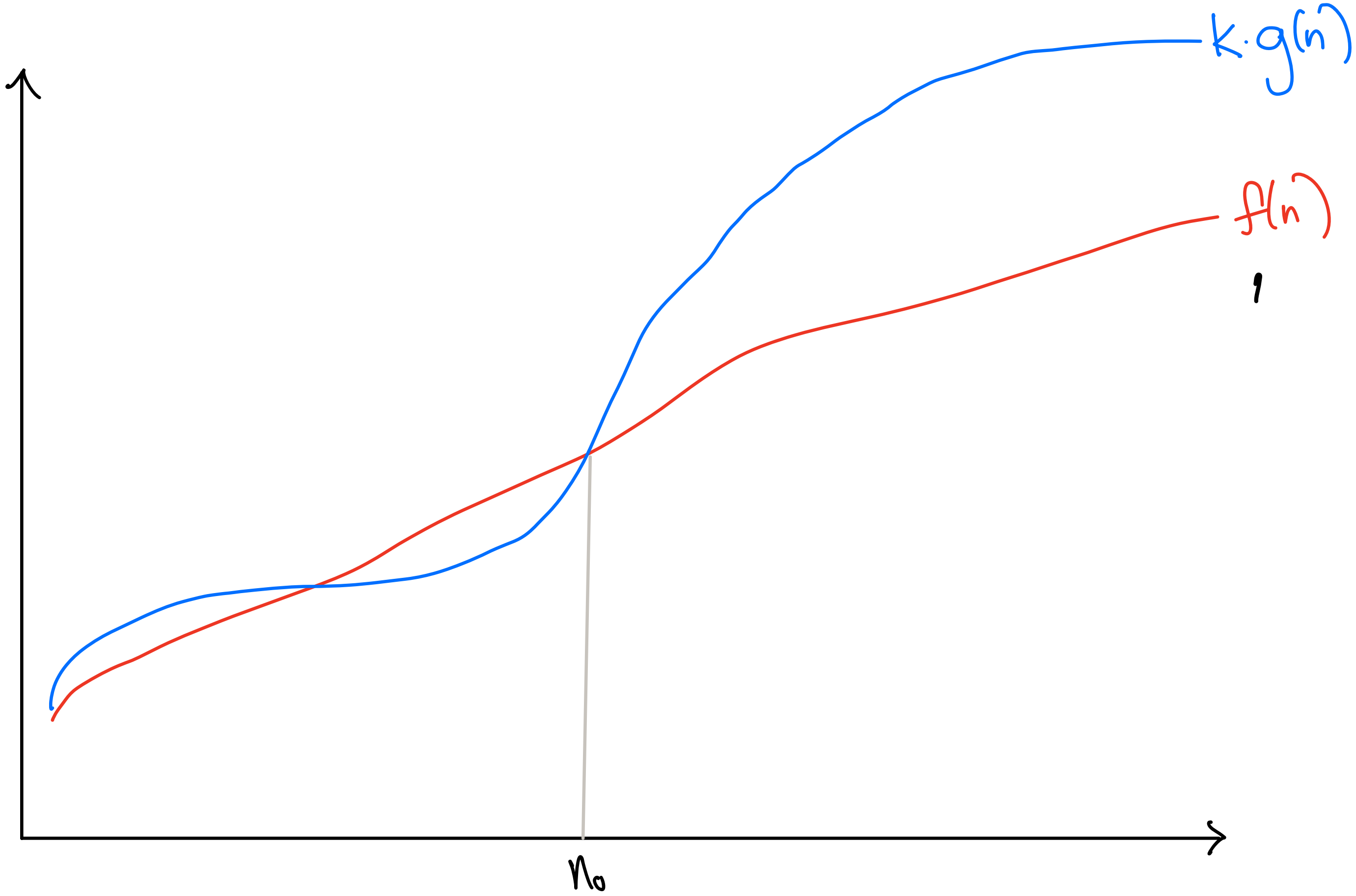
Big-O

$f = O(g)$ means

$$
\begin{aligned}
&\exists k \in \mathbb{R}_{>0}(\\
&\qquad \exists n_0 \in \mathbb{N}.(\\
&\qquad\qquad \forall n \in \mathbb{N}.(n > n_0 \implies \\
&\qquad\qquad\qquad f(n) \leq k \cdot g(n) \\
&\qquad\qquad) \\
&\qquad) \\
&).
\end{aligned}
$$

Less formally, f is **at most** $kg(n)$ for large enough inputs, where k is some constant.

Big-O



In the following slides, the differences in the formal definitions from Big-O are highlighted in red.

Big-Omega

$f = \Omega(g)$ means

$$
\begin{aligned}
&\exists k \in \mathbb{R}_{>0}(\\
&\qquad \exists n_0 \in \mathbb{N}.(\\
&\qquad\qquad \forall n \in \mathbb{N}.(n > n_0 \implies \\
&\qquad\qquad\qquad f(n) \geq k \cdot g(n) \\
&\qquad\qquad) \\
&\qquad) \\
&).
\end{aligned}
$$

Less formally, f is **at least** $kg(n)$ for large enough inputs, where k is some constant.

Big-Theta

$f = \Theta(g)$ means

$$
\begin{aligned}
&\exists k_1, k_2 \in \mathbb{R}_{>0}(\\
&\qquad \exists n_0 \in \mathbb{N}.(\\
&\qquad\qquad \forall n \in \mathbb{N}.(n > n_0 \implies \\
&\qquad\qquad\qquad k_1 \cdot g(n) \leq f(n) \leq k_2 \cdot g(n) \\
&\qquad\qquad) \\
&\qquad) \\
&).
\end{aligned}
$$

Less formally, f is **between** $k_1 \cdot g(n)$ and $k_2 \cdot g(n)$ for large enough inputs, where k_1 and k_2 are some constants.

Equivalently, $f = O(g)$ and $f = \Omega(g)$.

Little-o

$f = o(g)$ if

$$
\begin{aligned}
&\forall k \in \mathbb{R}_{>0}(\\
&\quad \exists n_0 \in \mathbb{N}.(\\
&\quad\quad \forall n \in \mathbb{N}.(n > n_0 \implies \\
&\quad\quad\quad f(n) < k \cdot g(n) \\
&\quad\quad) \\
&\quad) \\
&).
\end{aligned}
$$

Less formally, no matter how small a constant k I multiply g by, for all large enough inputs, $f(n)$ is less than $g(n)$.

Little-omega

$f = \omega(g)$ if

$$
\begin{aligned}
&\forall k \in \mathbb{R}_{>0}(\\
&\qquad \exists n_0 \in \mathbb{N}.(\\
&\qquad\qquad \forall n \in \mathbb{N}.(n > n_0 \implies \\
&\qquad\qquad\qquad f(n) > k \cdot g(n) \\
&\qquad\qquad) \\
&\qquad) \\
&).
\end{aligned}
$$

Less formally, no matter how large a constant k I multiply g by, for all large enough inputs, $f(n)$ is greater than $g(n)$.

A note about the definitions

There is some flexibility in these definitions. I.e. You can replace $<$ with $\leq$ and $>$ with $\geq$ (and vice versa) wherever your want.

You can also change the side the constant k is multiplied on if you want. I.e. multiply k to f instead of g.

It's a good exercise to prove this.

Asymptotics and orders

We can think of these asymptotics relations as

- $f = o(g)$ is like $f < g$
- $f = O(g)$ is like $f \leq g$
- $f = \Theta(g)$ is like $f \approx g$
- $f = \Omega(g)$ is like $f \geq g$
- $f = \omega(g)$ is like $f > g$

We'll sometimes use $\boxed{\prec, \preceq, \approx, \succeq, \succ}$ for $o, O, \Theta, \Omega, \omega$ respectively.

$n = O(n)$

$n \neq o(n)$

$n \not< n$

$n \leq n$

$f \leq g$

$g \geq f$

$f = O(g)$.

$g = \Omega(f)$.

Logs in this class

log in this class is always $\log_2$ unless otherwise specified. It is the true inverse of the the function that maps $x \mapsto 2^x$. I.e., for any $x \in \mathbb{R}$.

$$\log(2^x) = x,$$

and for any $y \in \mathbb{R}_{>0}$

$$2^{\log(y)} = y.$$

Fast Rules

$\prec$ o

$n!$

$$1 \prec \log(n) \prec n^{0.001} \prec n \prec n\log(n) \prec n^{1.001} \prec n^{1000} \prec 1.001^n \prec 2^n \prec n!$$

Helpful alternative definition for little-o if you know limits:

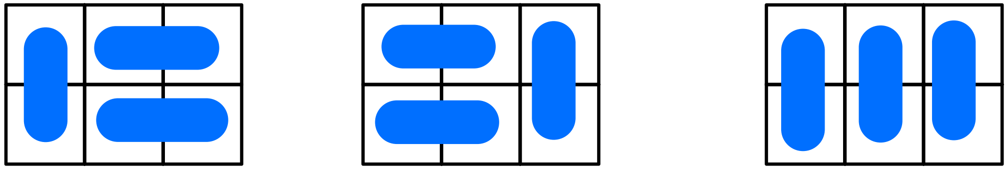
$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0 \iff f \prec g$$

Dominoes

How many ways are there to tile a $2 \times n$ grid using 2×1 dominoes?

Examples

2×3.

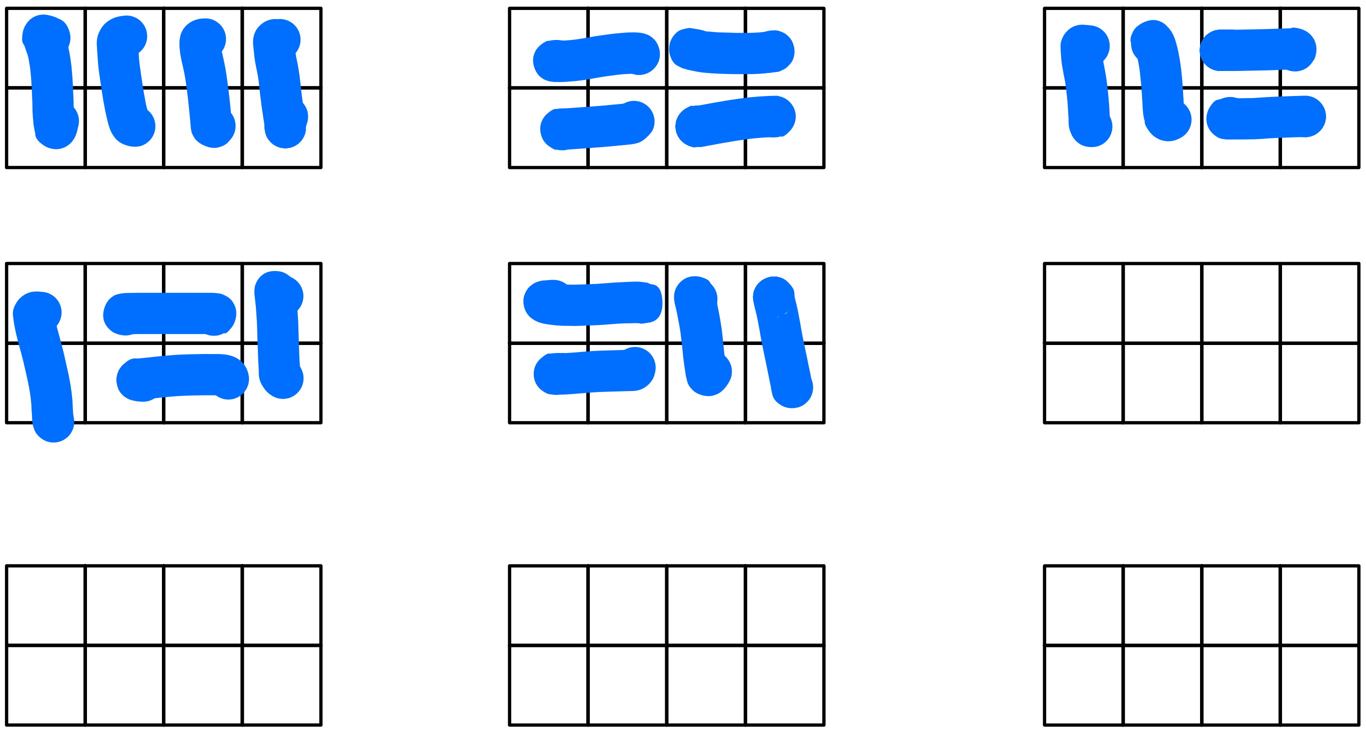


$T(3) = 3$

Number of tilings

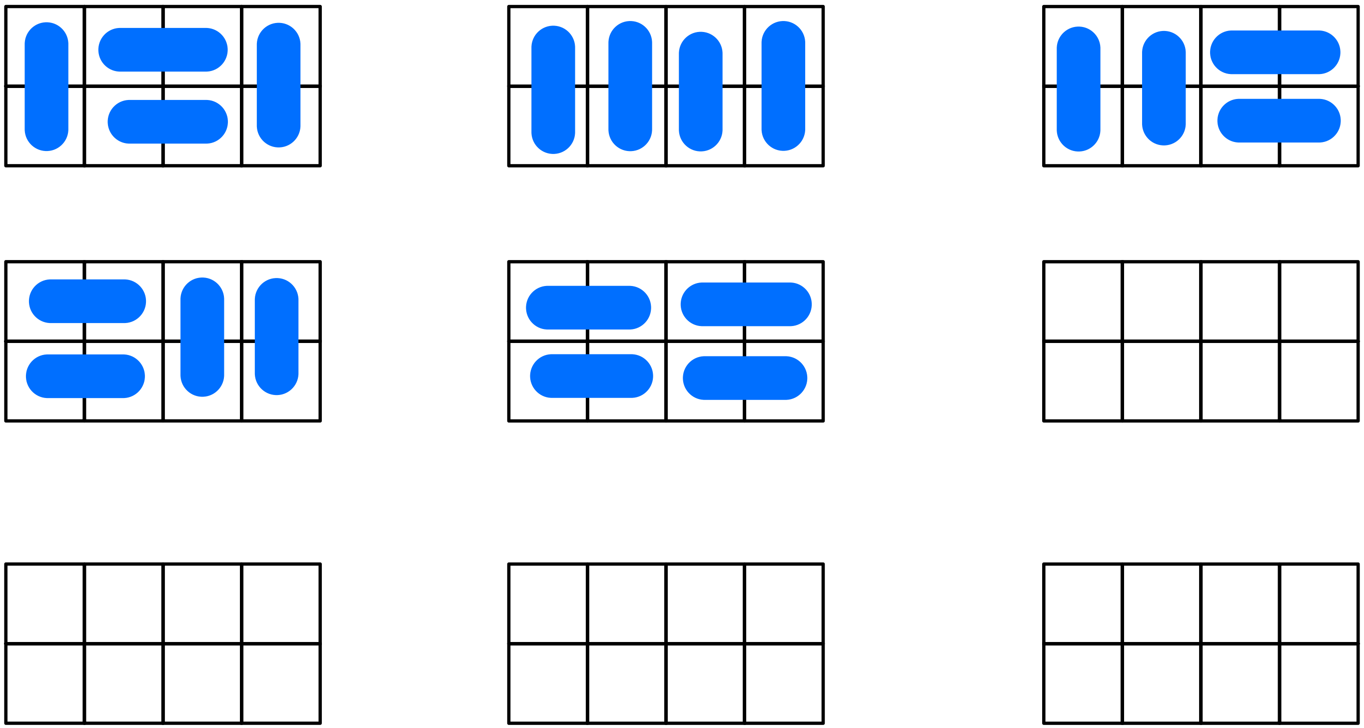
Let $T(n)$ be the number of tilings of a $2 \times n$ grid using 2×1 dominoes.

$T(4)$



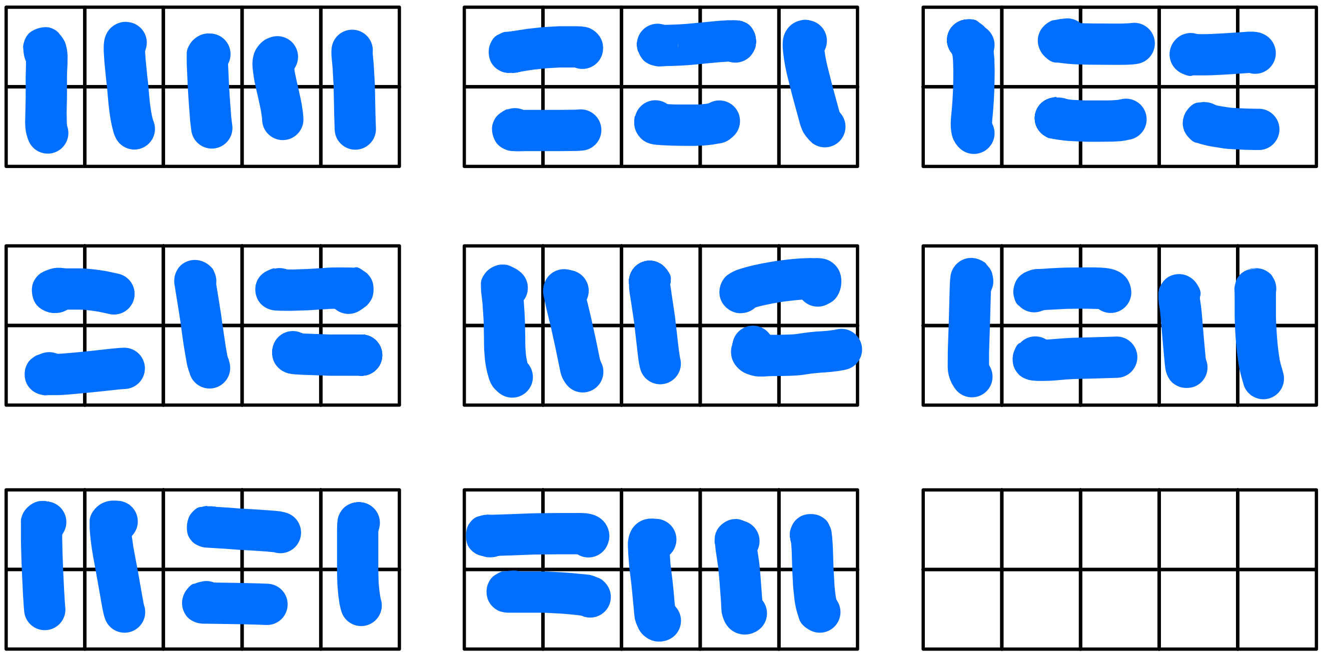
$T(4) = 5$

$T(4)$

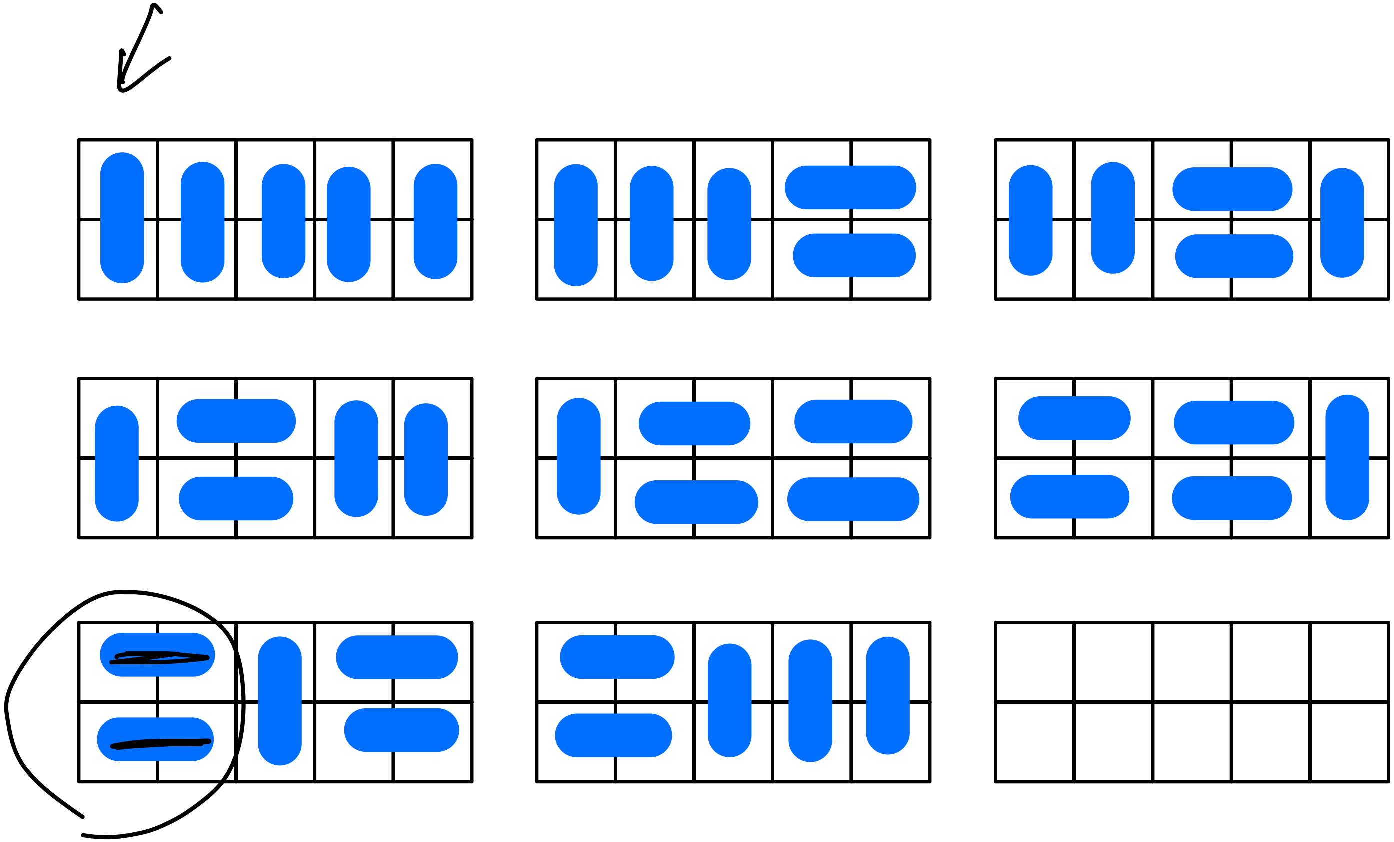


$T(4) = 5$

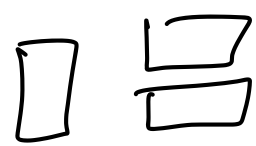
$T(5)$



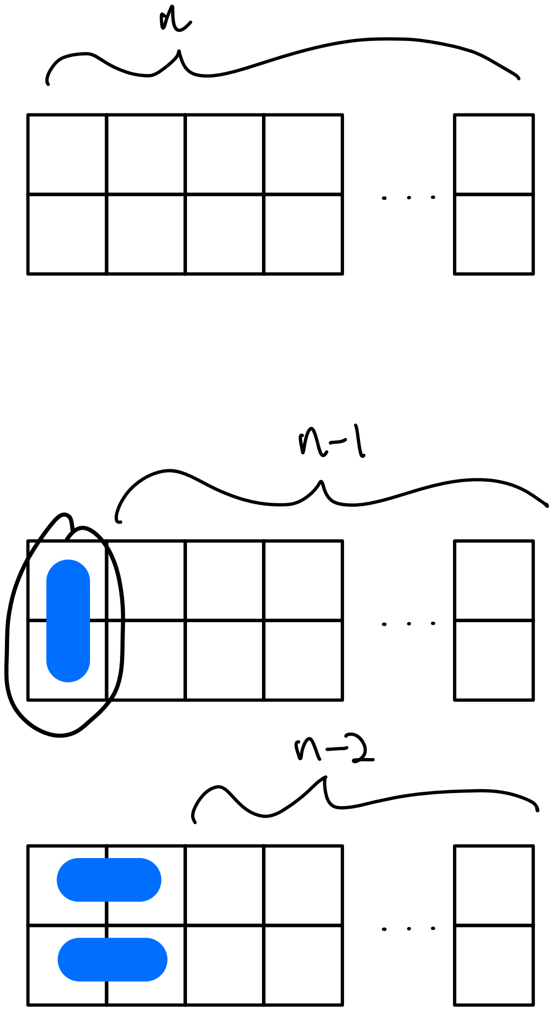
$T(5)$



$T(5) = 8$



Number of tilings - Recursively



Every tiling of $2 \times n$ is either

1. followed by a tiling of $2 \times (n-1)$

2. followed by a tiling of $2 \times (n-2)$!

$$\Rightarrow \underline{T(n) = T(n-1) + T(n-2)}.$$

Fibonacci Numbers

$$\mathrm{Fib}(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ \mathrm{Fib}(n-1) + \mathrm{Fib}(n-2) & n \geq 2 \end{cases}$$

We'll now study the asymptotics of $\mathrm{Fib}(n)$.

Note that $T(n)$ and $\mathrm{Fib}(n)$ have the same recursive relation. $T(n)$ are just the Fibonacci numbers shifted by one. I.e. $T(n) = \mathrm{Fib}(n+1)$.

An upper bound

Claim. $\forall n \in \mathbb{N}.(F(n) \leq 2^n)$

Base case: $F(0) = 0 \leq 1 = 2^0$

$F(1) = 1 \leq 2 = 2^1$

Inductive step: $F(k) = F(k-1) + F(k-2)$.

$\leq 2^{k-1} + 2^{k-2}$

$\leq 2^{k-1} + 2^{k-1}$ lost a lot!

$= 2 \cdot 2^{k-1}$

$= 2^k$.

$\blacksquare$

Tightening the analysis

$\boxed{\textbf{Claim. } \forall n \in \mathbb{N}.(F(n) \leq 2^n)}$

Tightening the analysis

Claim: $\text{Fib}(n) \le 1.8^n$

Base case: $\text{Fib}(0) = 0 \le 1 = 1.8^0$

$\text{Fib}(1): 1 \le 1.8 = 1.8^1$

Ind.

$$\text{Fib}(k) = \text{Fib}(k-1) + \text{Fib}(k-2)$$

$$\le 1.8^{k-1} + 1.8^{k-2}$$

$$\le 1.8 \cdot 1.8^{k-2} + 1.8^{k-2}$$

$$= 1.8^{k-2}(1.8 + 1)$$

$$\le 1.8^k.$$

Tightening the analysis

Claim: $F(n) \le 1.5^n$

Inductive step:

$$F(k+1) = F(k) + F(k-1)$$

$$\le 1.5^k + 1.5^{k-1}$$

$$= 1.5 \cdot 1.5^{k-1} + 1.5^{k-1}$$

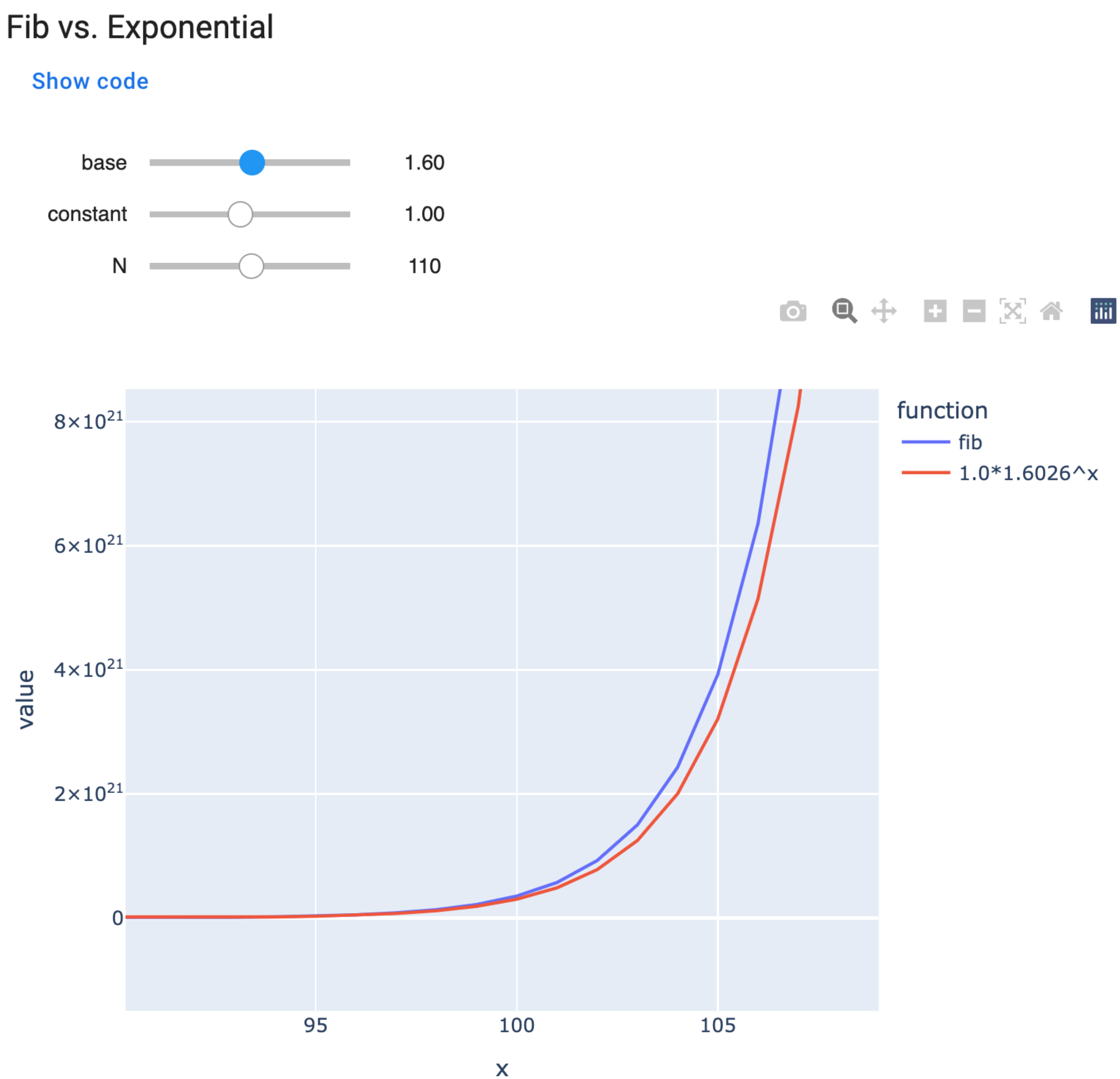
$$= 1.5^{k-1}(2.5)$$

$$\le 1.5^{k+1} \; ?$$

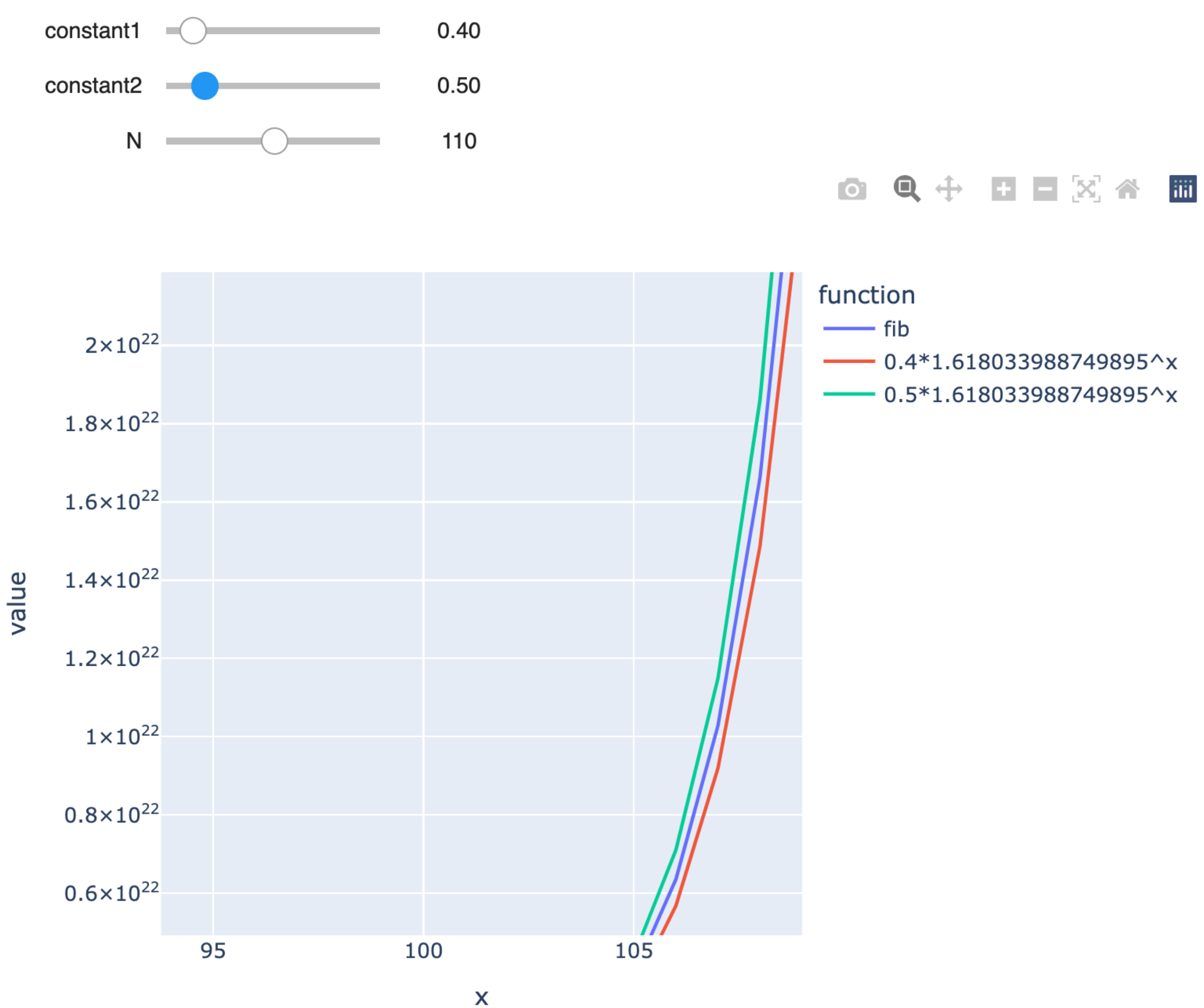
$1.5^2 = 2.25 \not\le$

Fibonacci - Code!

Fibonacci - Code!



Fibonacci - Code!

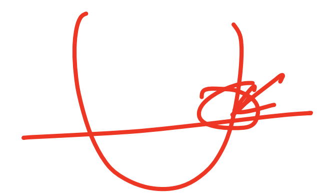


Optimizing the base

Claim: $F(n) \le x^n$

Inductive step:

$$
\begin{aligned}
F(k+1) &= F(k) + F(k-1) \\
&\le x^k + x^{k-1} \\
&= x^{k-1}\underbrace{(x+1)}. \\
&\;\vdots \\
&\le x^{k+1}.
\end{aligned}
$$



$\Rightarrow x+1 \le x^2.$

$x^2 - x - 1 \ge 0 \longrightarrow \dfrac{1 \pm \sqrt{1+4}}{2}$

$x = \dfrac{1 \pm \sqrt{5}}{2}$

$x \ge \dfrac{1+\sqrt{5}}{2} = \boxed{1.618\ldots}$

$ax^2+bx+c=0$

Upper bound

Claim. $\forall n \in \mathbb{N}.(\mathrm{Fib}(n) \leq \varphi^n)$

$\varphi = \frac{1+\sqrt{5}}{2}$, $\varphi^2 = \varphi + 1$.

base. ✓

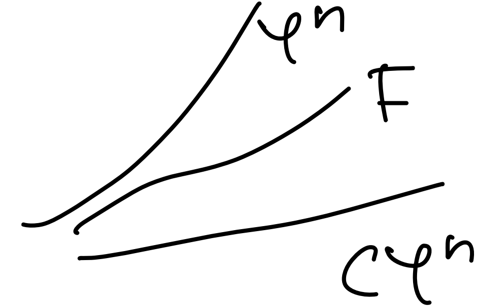
$$\mathrm{Fib}(k+1) = \mathrm{Fib}(k) + \mathrm{Fib}(k-1)$$

$$\leq \varphi^k + \varphi^{k-1}$$

$$= \varphi^{k-1}(\varphi + 1)$$

$$= \varphi^{k+1}$$

A lower bound $Fib(n) = \Omega(\varphi^n)$



Claim: $Fib(n) \ge c\,\varphi^n \quad \forall n \ge 1$

Base case: $F(1) = 1 \ge c \cdot \varphi$.

$F(2) = 1 \ge c \cdot \varphi^2$

$c = 0.0001$

$c \le 1/\varphi^2$.

Inductive

$F(k+1) = F(k) + F(k-1)$

$\ge c\varphi^k + c\varphi^{k-1}$

$= c\varphi^{k-1}(\varphi + 1)$ b/c $\varphi + 1 = \varphi^2$

$= c \cdot \varphi^{k+1}$

Fibonacci

$$\mathrm{Fib}(n) = \Theta(\varphi^n).$$

The complete answer - Binet's formula

$$\mathrm{Fib}(n) = \frac{\varphi^n - (1-\varphi)^n}{\sqrt{5}}$$

The complete answer - Binet's formula

$1-\varphi.$

$$\mathrm{Fib}(n) = \frac{\varphi^n - \cancel{(1-\varphi)^n}}{\sqrt{5}}$$

Note that $1 - \varphi \approx -0.618$, so the $\boxed{(1-\varphi)^n}$ term goes to zero really quickly and becomes irrelevant.

In fact, since $|(1-\varphi)^n/\sqrt{5}|$ is always less than $1/2$, $\mathrm{Fib}(n)$ is just $\boxed{\frac{\varphi^n}{\sqrt{5}}}$ rounded to the nearest whole number!

See the suggestions on slide 63 for further reading on solving recurrences exactly.

Takeaway

You can show the asymptotics of recurrences by induction!

Recurrences

Asymptotics Review

Dominoes

Binary Search

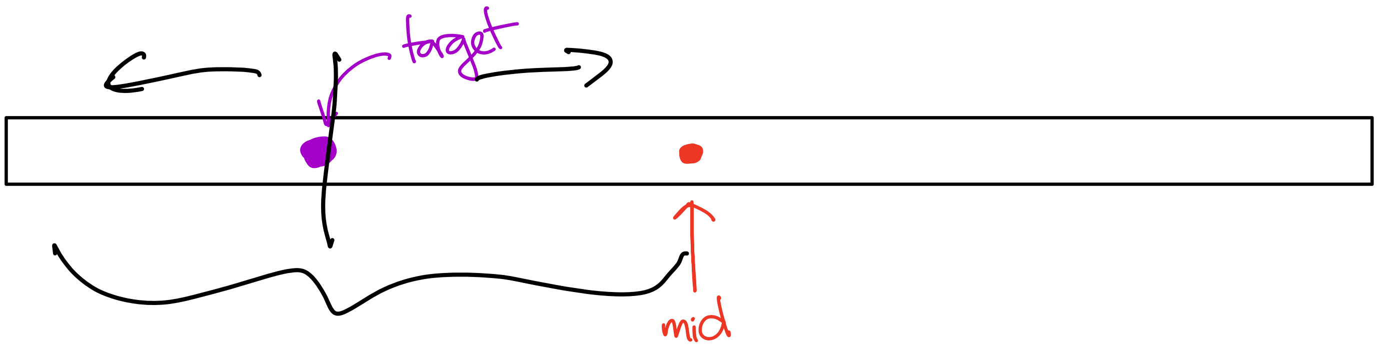
Searching in a sorted array

Inputs:

- A sorted list `l`
- A target value `target`

Output: The index of `target` in `l`. `None` if `target` is not in `l`.

Binary search intuition



if mid = target: we're done.

if mid < target, target is in the right half.

if mid > target, target is in the left half.

Binary Search - Code

Binary Search - Code

l[low : high]

```
def bin_search(l, target):
  return _bin_search(l, target, 0, len(l))

def _bin_search(l, target, low, high):
  if high == low:
    return None
  else:
    mid = (low + high)//2
    if l[mid] == target:
      return mid
    elif l[mid] < target:
      return _bin_search(l, target, mid+1, high)
    elif l[mid] > target:
      return _bin_search(l, target, low, mid)
```
[1]

[1] searches `l` between indices `low` inclusive and `high` **exclusive**

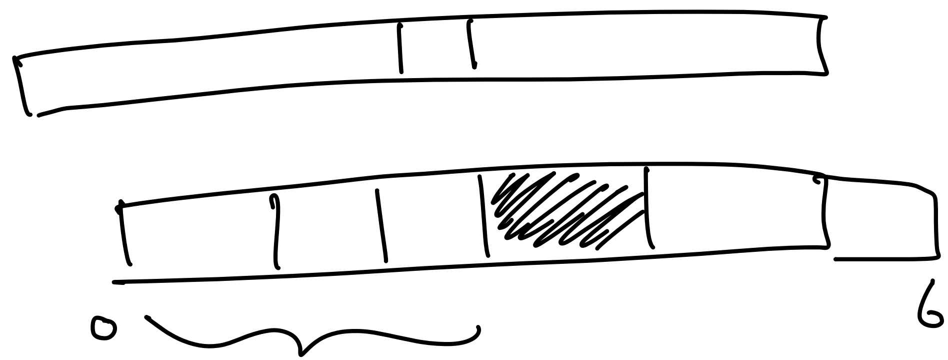
Binary Search - Code

```
bin_search_verbose(l, 26)
```

```
Sublist:  [6, 7, 17, 26, 28, 30, 33, 35, 40, 51, 58, 60, 68, 78, 78, 86, 88, 92, 95, 97]
Middle value:  58
Middle value is greater than the target - looking left.

Sublist:  [6, 7, 17, 26, 28, 30, 33, 35, 40, 51]
Middle value:  30
Middle value is greater than the target - looking left.

Sublist:  [6, 7, 17, 26, 28]
Middle value:  17
Middle value is less than the target - looking right.

Sublist:  [26, 28]
Middle value:  28
Middle value is greater than the target - looking left.

Sublist:  [26]
Middle value:  26
Middle value is equal to the target - done.
3
```

Binary Search

Let $T_{\text{BinSearch}}$ be the the function that maps the length of the input array to the worst case running time of the binary search algorithm. What is the recurrence for $T_{\text{BinSearch}}$?



$$T(n) = T\left(\left\lceil \frac{n-1}{2} \right\rceil\right) + 1$$

$$\left\lceil \frac{n-1}{2} \right\rceil$$

$$= \lfloor n/2 \rfloor$$

Binary Search

Let $T_{\text{BinSearch}}$ be the the function that maps the length of the input array to the worst case running time of the binary search algorithm. What is the recurrence for $T_{\text{BinSearch}}$?

Let's say doing a comparison and returning a value takes 1 unit of work (we could replace this with a constant amount of work c). The point is, the amount of work required to do these operations does not grow with the array length.

If $\boxed{n = 0}$ we return `None`, so $\boxed{T_{\text{BinSearch}}(0) = 1}$. In the recursive case, the value in the middle is not equal to the target. One side of the middle has $\lceil (n-1)/2 \rceil$ values and the other has $\lfloor (n-1)/2 \rfloor$. In the worst case, we call the algorithm recursively on a list of size $\lceil (n-1)/2 \rceil = \lfloor n/2 \rfloor$. Thus, for $n \geq 1$,

$$\boxed{T_{\text{BinSearch}}(n) = T_{\text{BinSearch}}(\lfloor n/2 \rfloor) + 1}$$

Some values

$T(n) = T(\lfloor n/2 \rfloor) + 1$

n	$T_{\text{BinSearch}}(n)$
0	1
1	
2	
3	
4	

Some values

$T(n) = T(\lfloor n/2 \rfloor) + 1$.

n	$T_{\text{BinSearch}}(n)$
0	1
1	2
2	3
3	3
4	4
5	4
6	4
7	4
8	5
15	5
16	6

$T(n) \leq n+1$

An upper bound

$$\boxed{T_{\text{BinSearch}}(n) = T_{\text{BinSearch}}(\lfloor n/2 \rfloor) + 1}$$

Claim. $T_{\text{BinSearch}}(n) = O(n)$. In particular, we claim for all $n \in \mathbb{N}. n \geq 1$, $T_{\text{BinSearch}}(n) \leq n + 1$.

Let $P(n)$ be $\boxed{T(n) \leq n+1}$

Base case $P(1)$: $T(1) = 2 \leq 1+1$. ✓

Inductive step Let $k \geq 2$. and assume $\underline{P(1) \ldots P(k-1)}$.

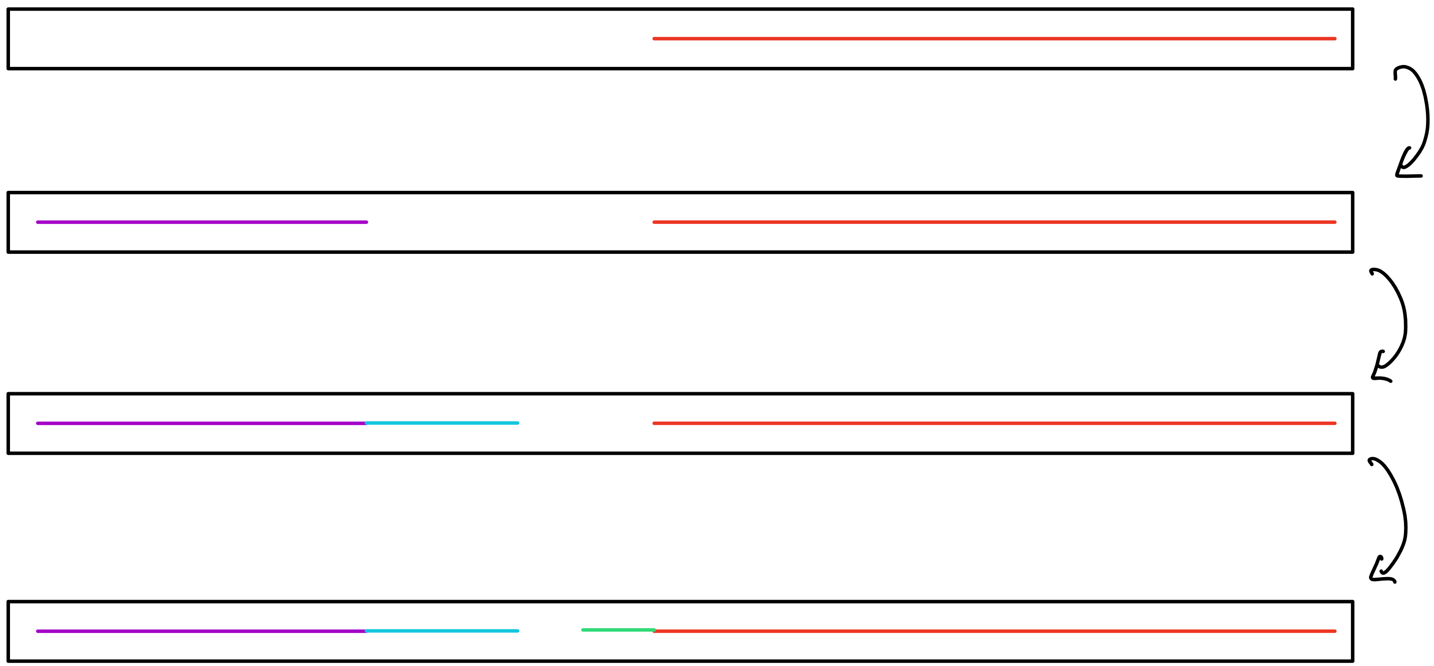
WTS $P(k)$. $T(k) = T(\lfloor k/2 \rfloor) + 1$

$\leq \lfloor k/2 \rfloor + 1 + 1$

$\leq (k-1) + 1 + 1 = k+1$

since $k \geq 2$, $\lfloor k/2 \rfloor \leq \underline{k-1}$.

What should the actual runtime be?



Everytime the array gets halved! $\Rightarrow$ if we start w/ 2^n elements we get to a single element in n steps.
if we started with n elements we need $\approx \log(n)$ steps.

$T_{\mathrm{BinSearch}}(n) = O(log(n))$

$$\boxed{T_{\mathrm{BinSearch}}(n) = T_{\mathrm{BinSearch}}(\lfloor n/2 \rfloor) + 1}$$

Claim: $T_{\mathrm{BinSearch}}(n) = O(\log(n))$. In particular, for all $n \geq 1$, $T_{\mathrm{BinSearch}}(n) \leq c \log(n) + d$ where c and d are constants that we will pick later.

A better upper bound

$$T_{\text{BinSearch}}(n) = T_{\text{BinSearch}}(\lfloor n/2 \rfloor) + 1$$

$T(n) \leq c\log(n) + d. \quad \forall n \geq 1.$ 0

Base case : $T(1) = 2 \leq \cancel{c\log(1)} + d.$

true as long as $d \geq 2$

Inductive step: let $k \in \mathbb{N}$, $k \geq 2$. assume $P(1), \ldots, P(k-1)$.

$$T(k) = T(\lfloor k/2 \rfloor) + 1.$$

$$\leq c\log(\lfloor k/2 \rfloor) + d + 1$$

$$\leq c\log(k/2) + d + 1.$$

$$= c(\log(k) - \log(2)) + d + 1$$

set $c = 1$

$$= c\log(k) - c + d + 1 \quad \ldots \leq c\log(k) + d.$$

Tips

Here are some tips for showing $T(n) = O(f(n))$

- Try proving $T(n) \leq cf(n) + d$ for some numbers c and d. After running the proof go back and figure out what c and d need to be for your proof to work.
- Sometimes in the inductive step, you might find it helpful to assume k is larger that some constant for example, $k \geq 3$. If this is the case, show $\forall n \in \mathbb{N}, n \geq 3. T(n) \leq f(n)$, and change the base case! (This is like the $n^2 \leq 2^n$ example from two lectures ago where we used the assumption that $k \geq 4$ in the inductive step.)

The exact answer

n	$T_{\mathrm{BinSearch}}(n)$
1	2
2	3
3	3
4	4
5	4
6	4
7	4
8	5
15	5
16	6

Write any $n \in \mathbb{N}$ with $n \geq 1$ as $2^i + x$ for some $i \in \mathbb{N}, x \in \mathbb{N}$ with $x \leq 2^i - 1$.

Claim. $\forall i \in \mathbb{N}, \forall x \in \mathbb{N}, x \leq 2^i - 1.(T_{\mathrm{BinSearch}}(2^i + x) = i + 2)$

$$T_{\mathrm{BinSearch}}(2^i + x) = i + 2$$

Getting a good guess

Making a good guess is important in solving recurrences by induction. We'll see a method to do this next week.

Additional Notes

If you want a general method for fully solving recurrences, you'll need to study generating functions and partial fraction decomposition. See chapter 7 of *Concrete Mathematics* by Don Knuth for an excellent introduction. Or take a class on combinatorics.

Announcement: Tutorial Change

If you had tutorial with Logan in BA2159, please go to the following room instead for the rest of the semester.

- If your birthday is before July 2nd, go to BA2195.
- Else, go to BA2139.